Информатика в техническом университете

Г.С.Иванова, Т.Н.Ничушкина, Е.К.Пугачев

Объектно-ориентированное программирование

TObject

TMyFigure

TMyCircle

TMySquare

Издательство МГТУ имени Н.Э.Баумана

Информатика в техническом университете

Информатика в техническом университете

Серия основана в 2000 году

РЕДАКЦИОННАЯ КОЛЛЕГИЯ:

д-р техн. наук *И.Б. Федоров* — главный редактор
д-р техн. наук *А.А. Марков* — зам. главного редактора
д-р техн. наук *Ю.М. Смирнов* — зам. главного редактора
д-р техн. наук *В.Ф. Горнев*
д-р техн. наук *В.В. Девятков*
канд. техн. наук *И.П. Иванов*
д-р техн. наук *В.А. Матвеев*
д-р техн. наук *И.П. Норенков*
д-р техн. наук *В.В. Сюзев*
д-р техн. наук *Б.Г. Трусов*
д-р техн. наук *В.М. Черненький*
д-р техн. наук *В.А. Шахнов*

Г.С.Иванова, Т.Н.Ничушкина, Е.К.Пугачев

Объектно-ориентированное программирование

Под редакцией Г.С. Ивановой

Допущено Министерством образования
Российской Федерации
в качестве учебника для студентов
высших учебных заведений,
обучающихся по направлению
подготовки дипломированных специалистов
«Информатика и вычислительная техника»

Москва
Издательство МГТУ имени Н.Э.Баумана
2001

УДК 681.3.06(075.8)
ББК 32.973-018
И21

Рецензенты:
кафедра «Информатизация структур государственной службы»
Российской академии госслужбы при Президенте РФ
(зав. кафедрой профессор А.В. Петров);
профессор О.М. Брехов

Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К.

И21 Объектно-ориентированное программирование: Учеб. для вузов/ Под ред. Г.С. Ивановой. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001. – 320 с., ил. (Сер. Информатика в техническом университете).

ISBN 5–7038–1525–8

В учебнике рассмотрена технология объектно-ориентированного программирования (ООП). Приведены основные теоретические положения ООП и описание реализаций ООП в средах **Borland Pascal 7.0, С++ 3.1, Delphi и С++ Builder**. Подробно рассмотрена специфика создания Windows-приложений с использованием современных средств ООП.

Содержание учебника соответствует курсу лекций, который авторы читают в МГТУ им. Н.Э. Баумана.

Для студентов вузов, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника» и специальностям: «Вычислительные машины, системы, комплексы и сети», «Автоматизированные системы обработки информации и управления», «Программное обеспечение вычислительной техники и информационных систем». Может быть полезен всем изучающим объектно-ориентированное программирование.

УДК 681.3.06(075.8)
ББК 32.973-018

© Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев, 2001
© МГТУ им. Н.Э. Баумана, 2001
ISBN 5–7038–1525–8
© Издательство МГТУ им. Н.Э. Баумана, 2001

СОДЕРЖАНИЕ

Предисловие 7

1. Теоретические основы объектно-ориентированного программирования 11

1.1. От процедурного программирования к объектному 11
1.2. Основные принципы и этапы объектно-ориентированного программирования 18
1.3. Объектная декомпозиция 25
1.4. Объекты и сообщения 33
1.5. Классы 35
1.6. Основные средства разработки классов 39
1.7. Дополнительные средства и приемы разработки классов 48

2. Средства объектно-ориентированного программирования в Borland Pascal 7.0 59

2.1. Определение класса 59
2.2. Наследование 63
2.3. Полиморфизм 67
2.4. Динамические объекты 75
2.5. Создание библиотек классов 83
2.6. Композиция и наполнение 86
2.7. Разработка программ с использованием объектно-ориентированного программирования 92

3. Средства объектно-ориентированного программирования в Borland C++ 3.1 99

3.1. Определение класса 99
3.2. Конструкторы и деструкторы 110
3.3. Наследование 118

3.4. Полиморфизм 129
3.5. Дружественные функции и классы 135
3.6. Переопределение операций 138
3.7. Особенности работы с динамическими объектами 144
3.8. Параметризованные классы 154
3.9. Контейнеры 161

4. Создание приложений Windows 169

4.1. Семейство операционных систем Windows с точки зрения программиста 169
4.2. Структура приложения Windows 175
4.3. Технология разработки приложений Windows в средах Delphi и C++Builder 178

5. Объектная модель Delphi Pascal 192

5.1. Определение класса 192
5.2. Особенности реализации переопределения методов 202
5.3. Свойства 210
5.4. Метаклассы 219
5.5. Делегирование 228
5.6. Библиотека стандартных классов Delphi 233
5.7. Создание и обработка сообщений и событий 241
5.8. Обработка исключений 251

6. Объектная модель C++ Builder 260

6.1. Расширение базовой объектной модели C++ 260
6.2. Исключения 273
6.3. VCL-совместимые классы 283
6.4. Различие реализации объектных моделей C++, Delphi и C++Builder 294

Заключение 306

Список литературы 307

Предметный указатель 310

Перечень примеров 312

ПРЕДИСЛОВИЕ

Сложность современного программного обеспечения требует от разработчиков владения наиболее перспективными технологиями его создания. Одной из таких технологий на настоящий момент является объектно-ориентированное программирование (ООП). Применение ООП позволяет разрабатывать программное обеспечение повышенной сложности за счет улучшения его технологичности (лучших механизмов разделения данных, увеличения повторяемости кодов, использования стандартизованных интерфейсов пользователя и т.д.)

Однако ООП существенно отличается от традиционного программирования, к которому мы привыкаем со школьной скамьи, и потому считается трудно осваиваемым. Чтобы технологически грамотно использовать ООП, необходимо хорошо понимать его основные концепции и научиться мыслить при разработке программы в понятиях ООП. Для того чтобы помочь вам в этом, и написана данная книга.

Практика преподавания ООП показывает, что идеология ООП лучше осваивается начинающими программистами, поэтому чем раньше будущий программист ознакомится с этой технологией, тем лучше.

В основе книги лежат конспекты лекций, читаемых авторами в течение нескольких лет в МГТУ им. Н.Э. Баумана на кафедре «Компьютерные системы и сети» в цикле базовой подготовки по программированию, рассчитанной на студентов младших курсов. К началу чтения данного раздела студенты уже владеют основными навыками программирования на Паскале и С, а также работы в операционных системах MS DOS и Windows'95 (или старше). Аналогичная подготовка желательна и для читателей предлагаемой книги.

Изложение построено следующим образом.

В главе 1 подробно представлены основные концепции и механизмы ООП. Описание выполняется чисто теоретически, без учета особенностей конкретных реализаций объектных моделей в рассматриваемых языках программирования. При чтении этой главы очень важно понять: чем рассматриваемый подход отличается от традиционного, что подразумевается под объектной декомпозицией, чем объектная декомпозиция отличается от процедурной, и какие основные механизмы реализуются средствами объектно-ориентированных языков программирования.

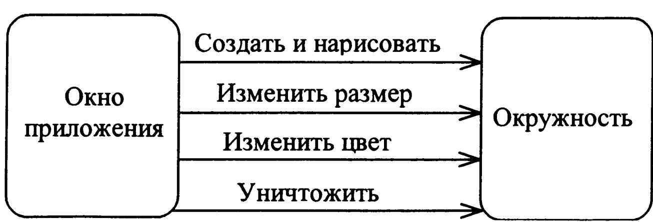
В главах 2 и 3 подробно рассмотрены средства ООП языков программирования, предназначенные для использования в среде MS DOS. Это упрощенная модель, предложенная в Borland Pascal 7.0 специально для облегчения «вживания» программистов в новую технологию, и практически полная объектная модель Borland C++ 3.1. Выбор именно этих языковых средств в значительной степени был обусловлен их широким распространением в нашей стране. Материал сопровождается большим количеством примеров, иллюстрирующих основные моменты изложения. При желании можно изучать одну из предложенных моделей, так как материал этих глав не связан, но в главе 2 много внимания уделяется выполнению декомпозиции, поэтому желательно ее хотя бы просмотреть.

Глава 4 содержит сведения об особенностях разработки приложений операционных систем Windows'95, Windows'98 и Windows NT, необходимые для понимания глав 5 и 6, посвященных объектным моделям, положенным в основу Delphi и C++ Builder. Разработка приложений Windows в этих средах существенно автоматизирована, что упрощает их освоение по сравнению с профессиональными средами типа Visual C++. В то же время эти среды предлагают разработчикам огромные библиотеки визуальных и невизуальных компонент и в том числе средства для связи с наиболее популярными базами данных, позволяя создавать в них достаточно серьезные программные продукты. Данная книга поможет вам сравнить обе модели, оценить их недостатки и достоинства.

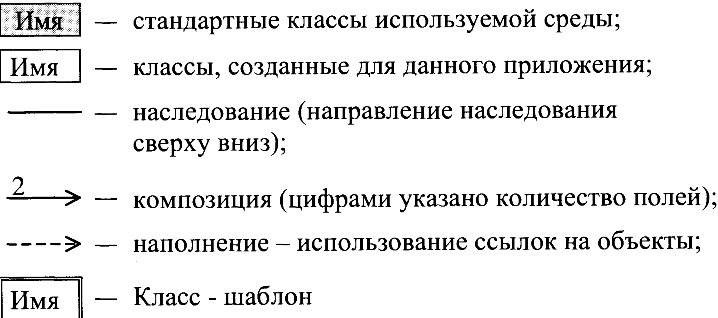
Серьезной проблемой при изложении материала было отсутствие единой принятой терминологии ООП. Так, например, традиционно в C++ для обозначения функций, включенных в класс, используется термин «компонентные функции», а в Pascal – «методы» и т.д. По мере возможности мы стремились использовать единую терминологию, оговаривая соответствующие особенности при изложении материала.

Материал данной книги содержит большое количество иллюстраций (структур объектов, классов), при их изображении использовались следующие обозначения.

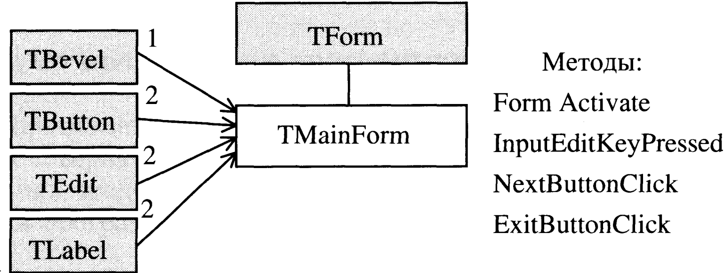
На структурной схеме программного продукта, которая при ООП является результатом объектной декомпозиции, объекты изображаются прямоугольниками (возможно со скругленными углами), а сообщения – линиями со стрелками, рядом с которыми указываются передаваемые сообщения. Например:



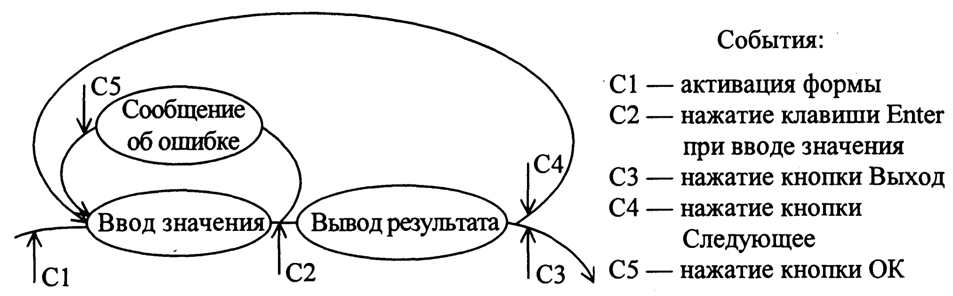
При изображении **диаграмм классов** использованы следующие условные обозначения:



В большинстве случаев рядом с разрабатываемым классом указываются названия полей и методов, которые предполагается включить в класс. Такое описание, как правило, позволяет сориентироваться в основных особенностях классов. Например:



При изображении **графа состояний интерфейса** сами состояния показываются в виде эллипсов, а возможные переходы – линиями со стрелкой на конце. На графе состояний интерфейса могут быть отмечены события, в результате наступления которых осуществляется тот или иной переход. Например:



Учебник содержит большое количество примеров программ на рассматриваемых языках программирования. Как правило, они иллюстрируют новые положения или сложные моменты.

Мы будем рады, если наша книга поможет вам в освоении ООП.

Главы 1, 4 – 6 написаны доцентом, канд. техн. наук Г.С. Ивановой; глава 2 – старшим преподавателем, канд. техн. наук Е.К. Пугачевым; глава 3 – доцентом, канд. техн. наук Т.Н. Ничушкиной.

Хочется выразить глубокую признательность рецензентам: кафедре «Информатизация структур государственной службы» Российской академии госслужбы при Президенте РФ (зав. кафедрой профессор А.В. Петров) и профессору О.М. Брехову (зав. кафедрой «ЭВМ, комплексы и сети», МАИ) за ценные замечания и рекомендации, которые были учтены авторами при работе над рукописью.

Авторы

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

ООП появилось в результате длительной эволюции технологии разработки программных продуктов В основу эволюции легло стремление ускорить процесс создания надежных программных средств. Объектные модели различных версий универсальных языков программирования имеют свои особенности, но в их основе лежат единые концепции, что позволяет говорить о теоретических основах ООП

1.1. От процедурного программирования к объектному

На протяжении всех лет существования практика программирования требовала совершенствования технологических приемов и создания на их основе таких средств программирования, которые упростили бы процесс разработки программ, позволяя создавать все более сложные программные системы.

Первые программы были организованы очень просто. Они состояли из собственно программы на машинном языке и обрабатываемых данных. Сложность программ ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение большого количества данных.

Создание сначала ассемблеров, а затем и языков высокого уровня сделало программу более обозримой за счет снижения уровня детализации и естественно позволило увеличить ее сложность.

Появление в языках средств, позволяющих оперировать подпрограммами, существенно снизило трудоемкость разработки программ. Подпрограммы можно было сохранять и использовать в других программах. В результате были накоплены огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы. Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.1).

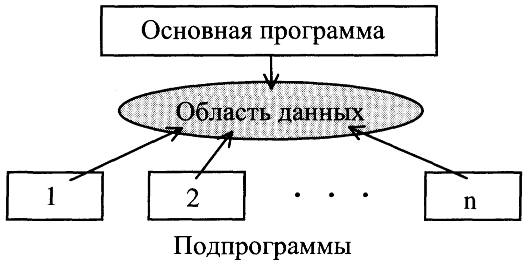


Рис. 1.1. Архитектура программы, использующей глобальную область данных

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, обычно подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала, полученное при последнем делении отрезка в ходе работы подпрограммы.

Необходимость исключения таких ошибок привела к идее использования в подпрограммах локальных данных (рис. 1.2).

И вновь сложность разрабатываемого программного обеспечения стала ограничиваться возможностью программиста отслеживать процессы обработки данных уже на новом уровне. К этому добавились проблемы согласования интерфейса при ведении разработки несколькими программистами. В результате встал вопрос создания технологии разработки сложных программных продуктов, снижающей вероятность появления ошибок.

Усилиями многих авторов такая технология была создана и получила название «структурное программирование» [3, 4].

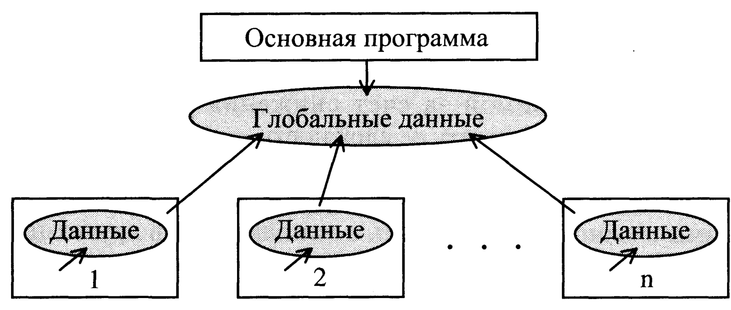


Рис. 1.2. Архитектура программы, использующей подпрограммы с локальными данными

Структурное программирование представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения.

Были сформулированы основные принципы выполнения разработки:

принцип нисходящей разработки, рекомендующий на всех этапах вначале определять наиболее общие моменты, а затем поэтапно выполнять детализацию (что позволяет последовательно концентрировать внимание на небольших фрагментах разработки);

собственно структурное программирование, рекомендующее определенные структуры алгоритмов и стиль программирования (чем нагляднее текст программы, тем меньше вероятность ошибки);

принцип сквозного структурного контроля, предполагающий проведение содержательного контроля всех этапов разработки (чем раньше обнаружена ошибка, тем проще ее исправить).

В основе структурного программирования лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40 ... 50 операторов) подпрограмм. В отличие от используемого ранее интуитивного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры, для получения которой рекомендовалось применять метод пошаговой детализации. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название *процедурной декомпозиции.*

Метод пошаговой детализации заключается в следующем:

определяется общая структура программы в виде одного из трех вариантов (рис. 1.3):

– последовательности подзадач (например, ввод данных, преобразование данных, вывод данных),

– альтернативы подзадач (например, добавление записей к файлу или их поиск),

– повторения подзадачи (например, циклически повторяемая обработка данных);

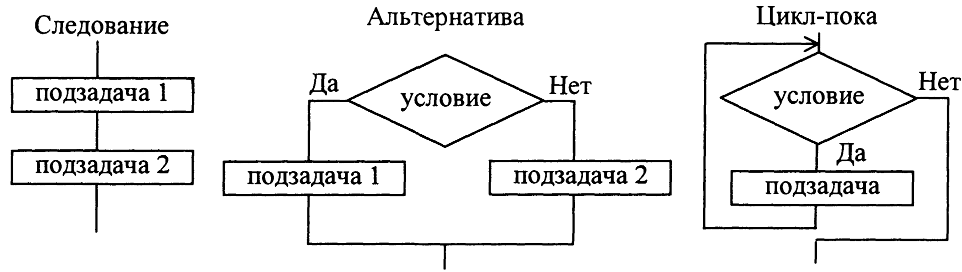


Рис. 1.3. Основные структуры процедурной декомпозиции

каждая подзадача, в свою очередь, разбивается на подзадачи с использованием тех же структур;

процесс продолжается, пока на очередном уровне не получается подзадача, которая достаточно просто реализуется средствами используемого языка (1 – 2 управляющих оператора языка).

Пример 1.1. Процедурная декомпозиция (программа «Записная книжка»). Пусть требуется разработать программу, которая в удобной для пользователя форме позволит записывать и затем находить телефоны различных людей и организаций. «Удобная» форма на современном уровне программирования предполагает общение программы с пользователем через «меню».

Анализ задачи показывает, что программу можно строить как последовательность подпрограмм. Следовательно, на первом шаге декомпозиции с использованием пошаговой детализации получаем

Основная программа:
Начать работу.
Вывести меню на экран.
Ввести команду.
Выполнить цикл обработки вводимых команд.
Завершить работу.

Первые три подзадачи, выявленные на данном шаге, представляются несложными, поэтому на следующем шаге детализируем действие «Выполнить цикл обработки вводимых команд»

Выполнить цикл обработки вводимых команд:
цикл-пока *команда ≠ «завершить работу»*
Выполнить команду.
Ввести команду
все-цикл.

После этого детализируем операцию «Выполнить команду». Выполняем декомпозицию, используя сразу несколько конструкций ветвления.

Выполнить команду:
если *команда= «открыть книжку»*
то *Открыть книжку*
иначе если *команда= «добавить»*
то *Добавить запись*
иначе если *команда= «найти»*
то *Найти запись*
все-если
все-если
все-если.

На этом шаге можно пока остановиться, так как оставшиеся действия достаточно просты. «Вложив» результаты пошаговой детализации, получим структурное представление алгоритма основной программы объемом не более 20 ... 30 операторов.

Основная программа:

```
Начать работу.
Вывести меню на экран.
Ввести команду.
цикл-пока команда ≠ «завершить работу»
  если команда= «открыть книжку»
   то Открыть книжку
    иначе  если команда= «добавить»
                то Добавить запись
                 иначе если команда= «найти»
                          то Найти запись
                        все-если
              все-если
все-если
Ввести команду
все-цикл
Завершить работу.
```

Примечание. Для записи алгоритма использован псевдокод, в котором следование отображается записью действий на разных строках, ветвление обозначается конструкцией **если** <условие> **то** <действие 1> **иначе** <действие 2>**все-если**, а цикл с проверкой условия выхода в начале – **цикл-пока** <действия> **все-цикл**. Вложенность конструкций определяется отступами от начала строки.

Окончательно, на первом уровне выделены подзадачи: «Вывести меню», «Ввести команду», «Открыть книжку», «Добавить запись» и «Найти запись».

На следующем уровне определяются подзадачи задач второго уровня, например:

Открыть_книжку:

```
Ввести имя  файла
если существует файл Имя_книжки
   то Открыть файл
   иначе Вывести сообщение об ошибке
все-если
```

На этом этапе получаем подзадачи: «Ввести имя файла» и «Открыть файл». Поступив аналогично с наиболее сложными подзадачами первого уровня, получаем схему двухуровневой алгоритмической декомпозиции задачи.

На рис. 1.4 показано, из каких подпрограмм будет состоять разрабатываемая система и взаимодействие последних по вызовам.

Созданная по результатам декомпозиции программа имеет правильную с точки зрения структурной технологии организацию, т.е. включает 12 относительно небольших подпрограмм, вызываемых из основной программы или подпрограмм более высокого уровня.

Сформулированная таким образом методика декомпозиции закрепила сложившийся в то время процедурный или алгоритмический подход к программированию, при котором основное внимание концентрируется на определении последовательности действий.

Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Как правило, они включали основные «структурные» операторы управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Примечание. Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития *структурирования данных* и, соответственно, в языках появляется возможность определения пользовательских типов данных. Одновременно усиливается стремление разграничить доступ к глобальным данным программы для уменьшения количества ошибок. Результатом было появление и развитие технологии модульного программирования.

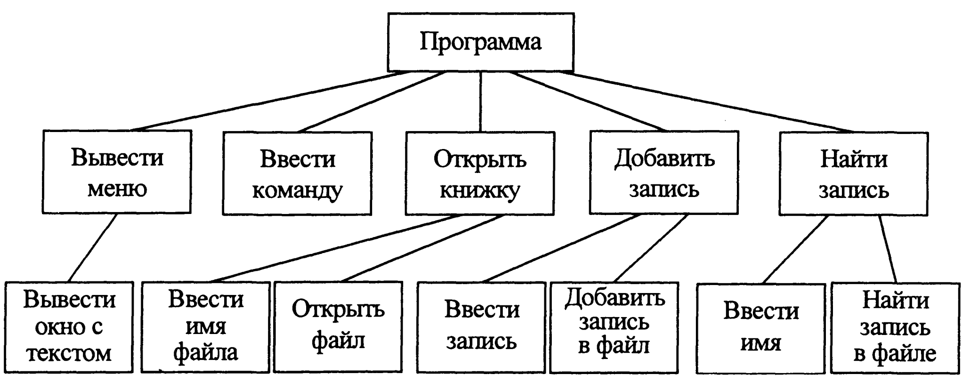


Рис. 1.4. Алгоритмическая декомпозиция системы «Записная книжка»

М о д у л ь н о е   п р о г р а м м и р о в а н и е (рис. 1.5) предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов, модуль подпрограмм вывода на принтер. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен.

Эту технологию поддерживают современные версии языков Pascal и С (С++), языки Ада и Modula.

Использование модульного программирования существенно упрощает ведение разработки программ несколькими программистами, каждый из которых разрабатывает свои модули. Внутренняя организация модулей скрыта от остальных и потому может изменяться независимо. Взаимодействие модулей осуществляется через специально оговоренные интерфейсы модулей. Кроме того, модули в дальнейшем могут использоваться в других разработках, что увеличивает производительность труда программистов.

Практика программирования показывает, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, *размер которых не превышает 100 000 операторов*. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы свыше 100 000

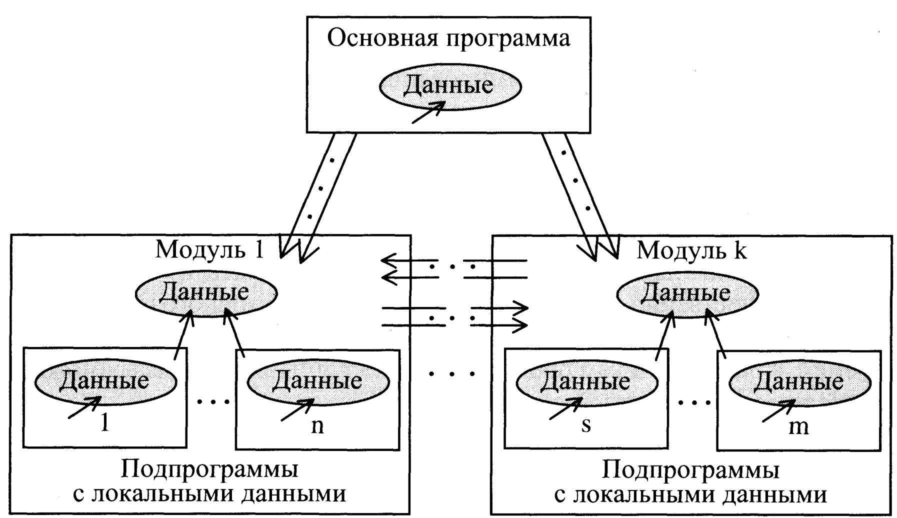


Рис. 1.5. Архитектура программы, состоящей из модулей

операторов обычно возрастает сложность межмодульных интерфейсов, и предусмотреть взаимовлияние отдельных частей программы становится практически невозможно.

Стремление уменьшить количество связей между отдельными частями программы привело к появлению *объектно-ориентированного программирования* (ООП).

1.2. Основные принципы и этапы объектно-ориентированного программирования

В теории программирования ООП определяется как технология создания сложного программного обеспечения, которая основана на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с *наследованием свойств* [2].

Взаимодействие программных объектов в такой системе осуществляется путем передачи *сообщений* (рис. 1.6).

Примечание. Такое представление программы впервые было использовано в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования – языке Smalltalk (70-е годы), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Ада, Modula.

Основное достоинство ООП – сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями, по сравнению с модульным программированием. Это достигается за счет более полной локализации данных и интегрирования их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы.

Кроме этого, объектный подход предлагает новые технологические средства разработки, такие как наследование, полиморфизм, композиция, наполнение, позволяющие конструировать сложные объекты из более простых. В результате существенно увеличивается показатель повторного использования кодов, появляется возможность создания библиотек объектов для различных применений, и разработчикам предоставляются дополнительные возможности создания систем повышенной сложности.

Основной недостаток ООП – некоторое снижение быстродействия за счет более сложной организации программной системы.

В основу ООП положены следующие *принципы*: абстрагирование, ограничение доступа, модульность, иерархичность, типизация, параллелизм, устойчивость.

Рассмотрим, что представляет собой каждый принцип.

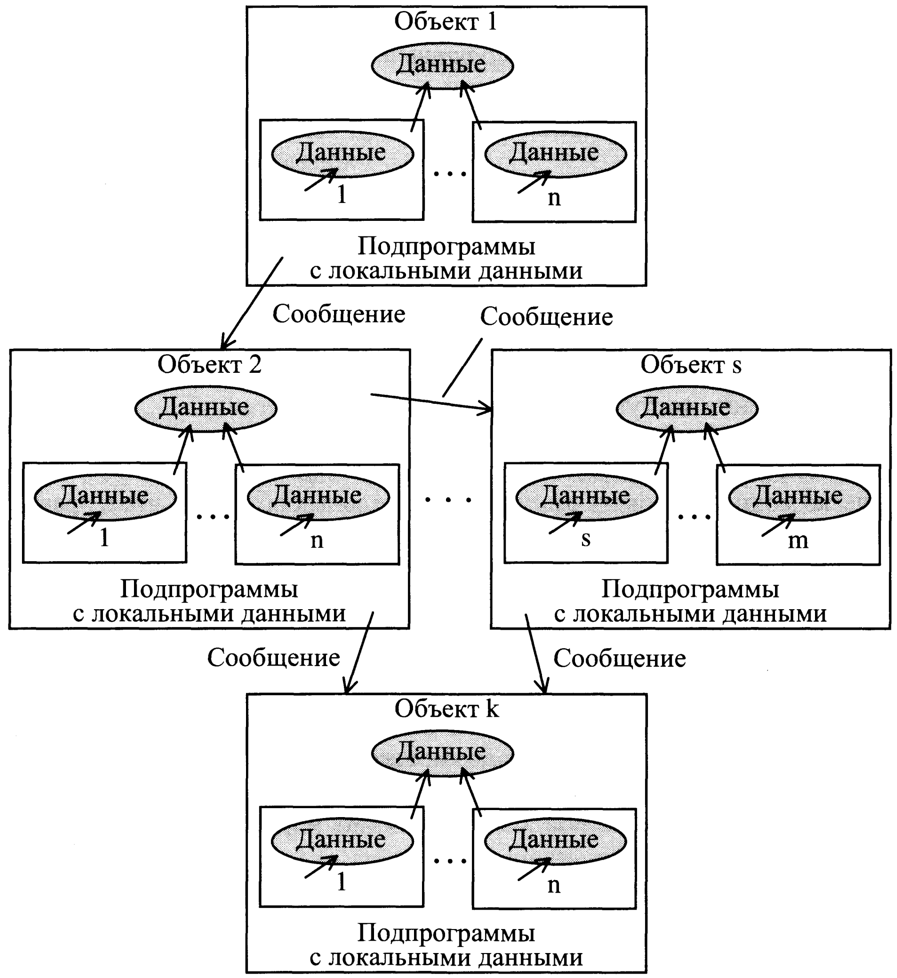


Рис. 1.6. Архитектура программы при ООП

Абстрагирование – процесс выделения абстракций в предметной области задачи. *Абстракция* – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. В соответствии с определением применяемая абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом вес, в третьем – материалы, из которых он сделан, в четвертом – закон движения предмета и т.д. Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния

анализируемого объекта, так и определяющих его поведение) в единую программную единицу некий *абстрактный тип* (класс).

Ограничение доступа – сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого.

Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:

интерфейс – совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);

реализация – совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения).

Ограничение доступа в ООП позволяет разработчику:

выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций;

легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

Сочетание объединения всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств получило название *инкапсуляции*.

Модульность – принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей (модулей). При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.

Иерархия – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем.

В ООП используются два вида иерархии.

Иерархия «целое/часть» – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне – при декомпозиции предметной области на объекты, на физическом уровне – при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе).

Иерархия «общее/частное» – показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Один из важнейших механизмов ООП – наследование свойств в иерархии общее/частное. *Наследование* – такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).

Типизация – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены). В языках с жесткой типизацией для каждого программного объекта (переменной, подпрограммы, параметра и т. д.) объявляется тип, который определяет множество операций над соответствующим программным объектом. Рассматриваемые далее языки программирования на основе Паскаля используют строгую, а на основе С – среднюю степень типизации.

Примечание. Интересно, что в С++ строгость типизация возрастает по сравнению с С, хотя ограничения типизации в этом языке можно почти полностью подавить.

Использование принципа типизации обеспечивает:

раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);

упрощение документирования;

возможность генерации более эффективного кода.

Тип может связываться с программным объектом статически (тип объекта определен на этапе компиляции – *раннее связывание*) и динамически (тип объекта определяется только во время выполнения программы – *позднее связывание*). Реализация позднего связывания в языке программирования позволяет создавать переменные – указатели на объекты, принадлежащие различным классам (*полиморфные объекты*), что существенно расширяет возможности языка.

Параллелизм – свойство нескольких абстракций одновременно находиться в активном состоянии, т.е. выполнять некоторые операции.

Существует целый ряд задач, решение которых требует одновременного выполнения некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами.

Реальный параллелизм достигается только при реализации задач такого типа на многопроцессорных системах, когда имеется возможность выполнения каждого процесса отдельным процессором. Системы с одним процессором имитируют параллелизм за счет разделения времени процессора между задачами управления различными процессами. В зависимости от типа используемой операционной системы (одно- или мультипрограммной)

разделение времени может выполняться либо разрабатываемой системой (как в MS DOS), либо используемой ОС (как в системах Windows).

Устойчивость – свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.

Различают:

временные объекты, хранящие промежуточные результаты некоторых действий, например вычислений;

локальные объекты, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до ее завершения;

глобальные объекты, существующие пока программа загружена в память;

сохраняемые объекты, данные которых хранятся в файлах внешней памяти между сеансами работы программы.

Все указанные выше принципы в той или иной степени реализованы в различных версиях объектно-ориентированных языков.

Объектно-ориентированные языки программирования. Язык считается объектно-ориентированным, если в нем реализованы первые четыре из рассмотренных семи принципов.

Примечание. Кроме этого, в теории программирования принято различать *объектно-ориентированные* и *объектные* языки программирования. Последние отличаются тем, что они не поддерживают наследования свойств в иерархии абстракций, например, Ада – объектный язык, а C++ и объектные версии Паскаля – объектно-ориентированные.

Несмотря на то, что принципиально ООП возможно на многих языках программирования, желательно для создания объектно-ориентированных программ использовать объектно-ориентированные языки, включающие специальные средства, например, Borland Pascal (начиная с версии 5.5), C++, Delphi и т.д.

В табл. 1.1 представлены сравнительные характеристики рассматриваемых в следующих главах языковых средств.

Самая простая объектная модель использована при разработке Borland Pascal 7.0. Она специально создавалась для облегчения перехода программистов на использование технологии ООП и не поддерживает абстракции метаклассов, почти не содержит специальных средств сокрытия реализации объектов. Но даже в таком варианте она позволяет создавать достаточно сложные системы. Объектные модели остальных языков являются практически полными.

Особое место занимают объектные модели Delphi и C++Builder. Эти модели обобщают опыт ООП для MS DOS и включают некоторые новые средства, обеспечивающие эффективное создание более сложных систем. На базе этих моделей созданы визуальные среды для разработки приложений Windows. Сложность программирования под Windows удалось существенно

снизить за счет создания специальных библиотек объектов, «спрятавших» многие элементы техники программирования.

Таблица 1.1. **Сравнительные характеристики моделей ООП в некоторых средах программирования**

Характеристики	Среды программирования			
	Borland Pascal 7.0	Borland C++ 3.1	Delphi	C++Builder
Абстракции:				
объекты	да	да	да	да
классы	да	да	да	да
Ограничение доступа:				
механизм сокрытия деталей реализации	внутри модуля	внутри класса и/или потомков класса	внутри модуля и потомков класса	внутри класса и/или потомков класса
обеспечение интерфейса к полям объекта	нет	нет	да	да
Модульность:				
интерфейс	да	да (файл-заголовок)	да	да (файл-заголовок)
реализация	да	да (файл)	да	да (файл)
Иерархичность:				
«целое – часть»	да	да	да	да
«общее – частное»	да	да	да	да
наследование	простое	множественное	простое	множественное
Типизация:				
степень	строгое	среднее	строгое	среднее
позднее связывание	да	да	да	да
шаблоны	нет	да	нет	да
метаклассы	нет	нет	да	нет
Параллелизм				
разделение времени	моделируется	моделируется	обеспечивается Windows	обеспечивается Windows
Устойчивость объектов	определяется временем жизни переменной			

Этапы разработки программных систем с использованием ООП. Процесс разработки программного обеспечения с использованием ООП включает четыре этапа: анализ, проектирование, эволюция, модификация.

Рассмотрим эти этапы.

А н а л и з. Цель анализа – максимально полное описание задачи. На этом этапе выполняется анализ предметной области задачи, объектная декомпозиция разрабатываемой системы и определяются важнейшие особенности поведения объектов (описание абстракций). По результатам анализа разрабатывается структурная схема программного продукта, на которой показываются основные объекты и сообщения, передаваемые между ними, а также выполняется описание абстракций.

П р о е к т и р о в а н и е. Различают:

логическое проектирование, при котором принимаемые решения практически не зависят от условий эксплуатации (операционной системы и используемого оборудования);

физическое проектирование, при котором приходится принимать во внимание указанные факторы.

Логическое проектирование заключается в разработке структуры классов: определяются поля для хранения составляющих состояния объектов и алгоритмы методов, реализующих аспекты поведения объектов. При этом используются рассмотренные выше приемы разработки классов (наследование, композиция, наполнение, полиморфизм и т.д.). Результатом является иерархия или диаграмма классов, отражающие взаимосвязь классов, и описание классов.

Физическое проектирование включает объединение описаний классов в модули, выбор схемы их подключения (статическая или динамическая компоновка), определение способов взаимодействия с оборудованием, с операционной системой и/или другим программным обеспечением (например, базами данных, сетевыми программами), обеспечение синхронизации процессов для систем параллельной обработки и т.д.

Э в о л ю ц и я   с и с т е м ы. Это процесс поэтапной реализации и подключения классов к проекту. Процесс начинается с создания основной программы или проекта будущего программного продукта. Затем реализуются и подключаются классы, так чтобы создать грубый, но, по возможности, работающий прототип будущей системы. Он тестируется и отлаживается. Например, таким прототипом может служить система, включающая реализацию основного интерфейса программного продукта (передача сообщений в отсутствующую пока часть системы не выполняется). В результате мы получаем работоспособный прототип продукта, который может быть, например, показан заказчику для уточнения требований. Затем к системе подключается следующая группа классов, например, связанная с реализацией некоторого пункта меню. Полученный вариант также тестируется и отлаживается, и так далее, до реализации всех возможностей системы.

Использование поэтапной реализации существенно упрощает тестирование и отладку программного продукта.

М о д и ф и к а ц и я. Это процесс добавления новых функциональных возможностей или изменение существующих свойств системы. Как правило, изменения затрагивают реализацию класса, оставляя без изменения его интерфейс, что при использовании ООП обычно обходится без особых неприятностей, так как процесс изменений затрагивает локальную область. Изменение интерфейса – также не очень сложная задача, но ее решение может повлечь за собой необходимость согласования процессов взаимодействия объектов, что потребует изменений в других классах программы. Однако сокращение количества параметров в интерфейсной части по сравнению с модульным программированием существенно облегчает и этот процесс.

Простота модификации позволяет сравнительно легко адаптировать программные системы к изменяющимся условиям эксплуатации, что увеличивает время жизни систем, на разработку которых затрачиваются огромные временные и материальные ресурсы.

Особенностью ООП является то, что объект или группа объектов могут разрабатываться отдельно, и, следовательно, их проектирование может находиться на различных этапах. Например, интерфейсные классы уже реализованы, а структура классов предметной области еще только уточняется. Обычно проектирование начинается, когда какой-либо фрагмент предметной области достаточно полно описан в процессе анализа.

Рассмотрение основных приемов объектного подхода начнем с объектной декомпозиции.

1.3. Объектная декомпозиция

Как уже упоминалось выше, при использовании технологии ООП решение представляется в виде *результата взаимодействия отдельных функциональных элементов* некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи.

В такой системе каждый функциональный элемент, получив некоторое входное воздействие (называемое *сообщением*) в процессе решения задачи, выполняет заранее определенные действия (например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и в свою очередь воздействовать на другие элементы). Процессом решения задачи управляет *последовательность сообщений*. Передавая эти сообщения от элемента к элементу, система выполняет необходимые действия.

Функциональные элементы системы, параметры и поведение которой определяются условием задачи, обладающие самостоятельным поведением (т.е. «умеющие» выполнять некоторые действия, зависящие от полученных сообщений и состояния элемента), получили название *объектов*.

Процесс представления предметной области задачи в виде совокупности

объектов, обменивающихся сообщениями, называется *объектной декомпозицией*.

Для того чтобы понять, о каких объектах и сообщениях идет речь при выполнении объектной декомпозиции в каждом конкретном случае, следует вспомнить, что первоначально объектный подход был предложен для разработки имитационных моделей поведения сложных систем. Набор объектов таких систем обычно определяется при анализе моделируемых процессов.

Пример 1.2. Объектная декомпозиция (имитационная модель бензоколонки). Пусть нас интересует зависимость длины очереди к бензоколонке от количества заправочных мест, параметров обслуживания каждого заправочного места и интенсивности поступления заявок на заправку топливом (рассматриваем топливо одного типа).

Задачи такого вида обычно решаются с использованием имитационных моделей. Модель программно имитирует реальный процесс с заданными параметрами, параллельно фиксируя его характеристики. Многократно повторяя процесс имитации с различными значениями параметров обслуживания или поступления заявок, исследователь получает конкретные значения характеристик, по которым строятся графики анализируемых зависимостей.

Процесс работы бензоколонки с тремя заправочными местами можно представить в виде диаграммы (рис. 1.7).

На этой диаграмме t1, t2, t3, ... – моменты времени, когда подъезжает очередная автомашина; Колонка 1 – Колонка 3 – заправочные места. Прямоугольник соответствует времени заправки автомашины (на

Рис. 1.7. Диаграмма обслуживания автомашин на бензоколонке

диаграммах колонок) или времени ожидания в очереди (на диаграмме очереди). Номера машин указаны внутри прямоугольников.

Первая подъехавшая автомашина занимает первую колонку, вторая – вторую, а третья – третью. К моменту появления четвертой автомашины освобождается первая колонка, и четвертая автомашина ее занимает. В момент появления пятой автомашины – все колонки заняты, она становится в очередь, ожидая освобождения колонки. Таким образом, автомашина, приехавшая на заправку, в зависимости от наличия свободных колонок, либо сразу подъезжает к колонке, либо ожидает в очереди освобождения колонки. Имитация данного процесса может выполняться следующим образом.

Процесс поступления автомашин на заправку будет имитироваться с помощью генератора заявок на обслуживание. Обычно в качестве генератора заявок используется датчик случайных чисел, работающий по заданному закону распределения. Фактически датчик случайных чисел будет определять интервал между приходами автомашин, используя который мы сможем определить время поступления следующей автомашины.

Процесс обслуживания будет имитироваться также с помощью датчика случайных чисел, который определяет время обслуживания каждой машины в момент занятия ею колонки. Таким образом, он определяет время освобождения колонки.

В цикле имитации модель бензоколонки будет опрашивать модель потока машин и блок колонок, какое событие произойдет раньше: придет следующая автомашина или освободится колонка. Определив время и тип будущего события, модель увеличит модельное время до момента наступления ближайшего события и инициирует его обработку, передав управление соответственно генератору потока машин или блоку бензоколонок.

Получив управление, модель генератора потока машин запросит у модели блока колонок, есть ли свободные колонки. Если незанятые колонки есть, то модель генератора потока машин сформулирует запрос на занятие одной из них, а если нет, то передаст модели очереди сообщение о постановке автомашины в очередь.

Обрабатывая событие освобождения колонки, модель блока колонок запросит у модели очереди информацию о наличии машин в очереди. Если машины в очереди есть, то модель блока колонок «заберет» одну машину из очереди и вновь займет колонку. Если машин в очереди нет, то она зафиксирует наличие свободной колонки.

Модель очереди, получая управление, будет отслеживать изменение размера очереди во времени. На основании этих данных при завершении моделирования можно определить среднюю длину очереди.

На рис. 1.8 представлена диаграмма объектов имитационной модели бензоколонки. На этой диаграмме показаны объекты и сообщения, которые эти объекты передают друг другу.

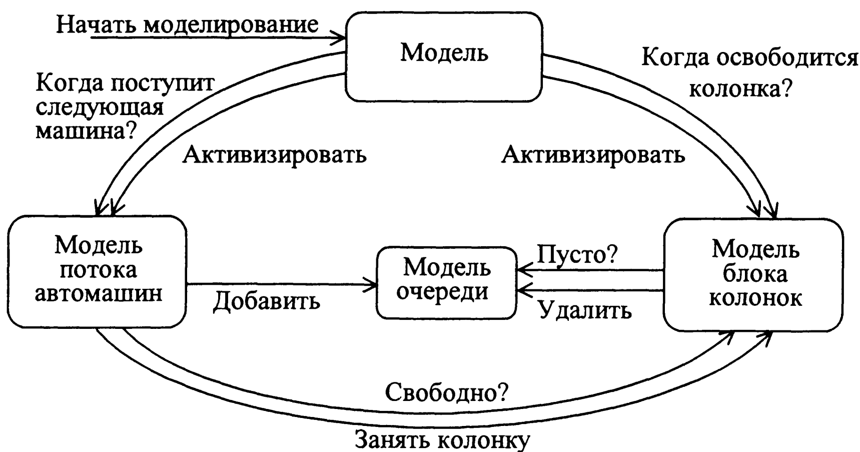


Рис. 1.8. Диаграмма объектов имитационной модели бензоколонки

Полученную модель можно реализовать в виде объектно-ориентированной программы. При программировании модели блока колонок, очереди и генератора потока автомашин будут представлены в виде объектов некоторых специально разработанных классов, а управляющая модель – в виде основной программы, инициирующей процесс моделирования. Передача сообщений в системе будет имитироваться как вызовы соответствующих методов объектов.

Объектная декомпозиция, так же как и процедурная, может применяться многократно, или быть многоуровневой. Это значит, что каждый объект может рассматриваться как система, которая состоит из элементов, взаимодействующих друг с другом через передачу сообщений. При многоуровневой декомпозиции на каждом уровне мы получаем объекты с более простым поведением, что позволяет разрабатывать системы повышенной сложности по частям.

Покажем, как происходит многоуровневая декомпозиция на том же примере. Для этого выполним декомпозицию объекта Блок колонок.

Пример 1.3. Декомпозиция объекта (Блок колонок). Модель блока колонок должна включать модели колонок и некоторый управляющий объект, который назовем Монитором. Монитор, получив сообщение, интерпретирует его и при необходимости генерирует сообщения моделям колонок. Например, получив сообщение-запрос о времени освобождения колонки, монитор отправит сообщения-запросы колонкам и выберет минимальное время из сообщенных колонками. Это минимальное время он вернет модели в качестве ответа на ее запрос. Получив сообщение о наступлении времени освобождения колонки, Монитор отправит соответствующие сообщения модели очереди и освобождаемой колонке (рис. 1.9).

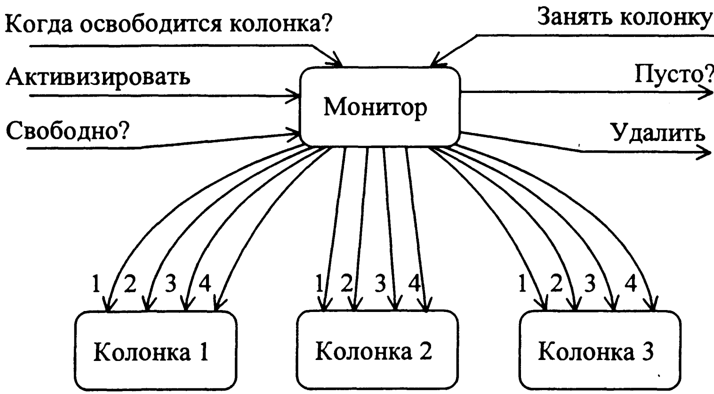


Рис. 1.9. Объектная декомпозиция Блока колонок:
1 – когда освободится колонка? 3 – колонка свободна?
2 – освободить колонку; 4 – занять колонку

Безусловно, это не единственный вариант декомпозиции объекта Блок колонок. При желании можно найти другие механизмы реализации того же поведения. Выбор конкретного варианта в каждом случае осуществляется разработчиком.

Таким образом, становится понятно, что в процессе объектной декомпозиции имитационных моделей выделяются части предметной области, которые будут *моделироваться как единое целое, обладающее собственным состоянием и поведением*, и определяется характер взаимодействия этих частей. Несколько сложнее дело обстоит с распространением идеи объектной декомпозиции на классы задач, напрямую не связанных с имитацией. По сути, в этом случае мы как бы программно *имитируем поведение* разрабатываемой системы.

Проиллюстрируем сначала на очень простом примере идею объектной декомпозиции для задач, не связанных с имитацией.

Пример 1.4. Простейший графический редактор. Выполним объектную декомпозицию программы, которая по запросу пользователя рисует одну из двух фигур: квадрат или круг. При желании пользователь должен иметь возможность изменить цвет контура, размер фигуры и координаты ее центра.

По правилам выполнения объектной декомпозиции разрабатывается имитационная модель программы. Для этого придется проанализировать все происходящие в имитируемой системе процессы и выделить элементы, обладающие собственным поведением, воздействующие на другие элементы и/или являющиеся объектами такого воздействия.

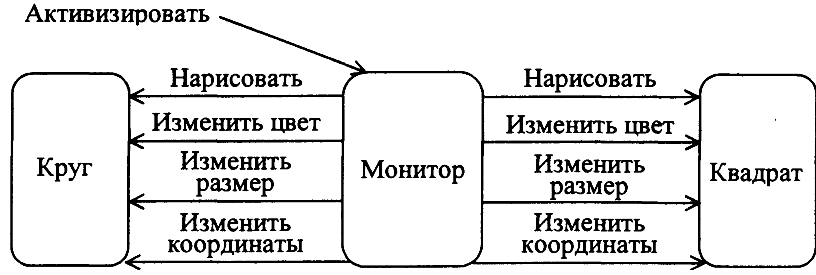


Рис. 1.10. Диаграмма объектов графического редактора

Основной процесс системы – процесс управления рисованием фигур,указанных пользователем. Все команды пользователя должны интерпретироваться, и в результате интерпретации должны формироваться команды на рисование или изменение параметров фигур. Эти процессы можно моделировать, используя три объекта: Монитор (блок управления, который получает и интерпретирует команды пользователя) и два объекта – фигуры (рис. 1.10), каждый со своими параметрами.

Фигуры получают следующие сообщения: «Нарисовать», «Изменить цвет контура», «Изменить размер», «Изменить координаты». Все эти сообщения инициируются Монитором в соответствии с командой пользователя. Получив от пользователя команду «Завершить», Монитор прекращает выполнение программы.

Примечание. На первый взгляд объектный подход по сравнению с процедурным кажется искусственным, но те преимущества, которые мы при этом получаем, безусловно, окупают затраченные усилия.

Пример 1.5. Объектная декомпозиция (программа «Записная книжка»). Попробуем выполнить объектную декомпозицию для программы «Записная книжка», на которой ранее демонстрировалась специфика процедурной декомпозиции (см. пример 1.1).

Как уже констатировалось ранее, работу с программами такого типа пользователю удобнее осуществлять через меню. Значит, при запуске программы на экране должно появиться меню, содержащее набор возможных операций. В зависимости от выбора пользователя далее будут активизироваться части программы, ответственные за открытие «книжки», добавление записей или за поиск записей (рис. 1.11).

На этом этапе анализа мы уже можем выделить четыре самостоятельных части программы, которые взаимодействуют в процессе ее выполнения: Меню, Открытие книжки, Ввод записей, Поиск записей. Часть Меню является управляющей и активизирует в процессе работы с программой остальные части для выполнения требуемых операций.

Добавим к системе еще несколько видимых объектов – сообщений пользователю, которые будут появляться на экране при обнаружении несоответствий в процессе работы, например, «Записная книжка не найдена» –

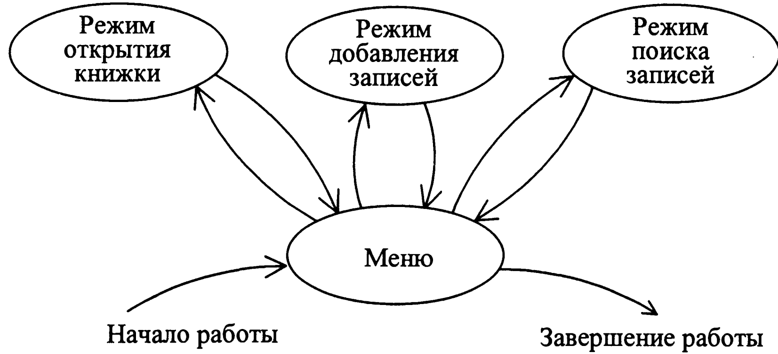


Рис. 1.11. Диаграмма состояний интерфейса пользователя (первый вариант)

активизируется Открытием книжки или «Нет информации об абоненте» – активизируется Поиском записей (рис. 1.12).

Объекты Меню, Открытие книжки, Ввод записей, Поиск записей в процессе работы должны получать информацию от пользователя и сообщать ему результаты работы, и соответственно должны иметь некоторое экранное представление. Совокупность таких экранных представлений (форм) образует интерфейс с пользователем.

Помимо объектов интерфейса система содержит еще, по крайней мере, один объект – Файл записей, используемый для хранения введенной информации. Этот объект должен получать от Открытия, Ввода и Поиска сообщения-команды соответственно открытия файла, добавления информации и поиска. Команда открытия должна сопровождаться именем файла, команда добавления информации – идентификационным заголовком и телефоном, а команда поиска – идентификационным заголовком.

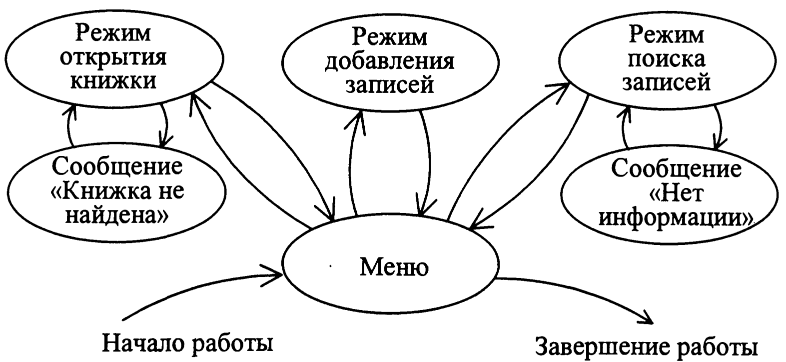


Рис. 1.12. Полная диаграмма состояний интерфейса пользователя

Объект Меню будет отвечать за выбор пунктов меню и вызов формы Открытие, формы Ввода и формы Поиска.

Форма Открытие должна осуществлять открытие или создание файла. Она будет вводить имя файла и вызывать объект Файл для выполнения операции открытия.

Форма Ввод должна вводить данные (фамилию, имя, отчество и телефон) и передавать эту информацию объекту Файл для сохранения.

Форма Поиск должна вводить данные для поиска (фамилию или имя, или и то и другое сразу) и запрашивать поиск телефона по введенным данным.

Все операции с файлом будет выполнять объект Файл.

Окончательный вариант объектной декомпозиции проектируемой системы представлен на рис. 1.13 (сравните с результатами процедурной декомпозиции для той же программы, приведенными на рис. 1.3).

Таким образом, можно сформулировать следующие рекомендации по выполнению объектной декомпозиции:

1. Для сложных систем объектная декомпозиция должна выполняться поэтапно: на первом этапе – объектная декомпозиция всей системы, на последующих – декомпозиция объектов как подсистем.

2. При декомпозиции системы в целом в качестве объектов могут выделяться элементы двух типов:

элементы интерфейса пользователя (окна меню, окна сообщений, окна форм ввода-вывода и т.д.);

средства хранения, организации и преобразования данных (базы данных, файлы, протоколы, структуры данных и т.д.).

При этом для каждого объекта должно определяться множество получаемых и передаваемых сообщений и основные характеристики.

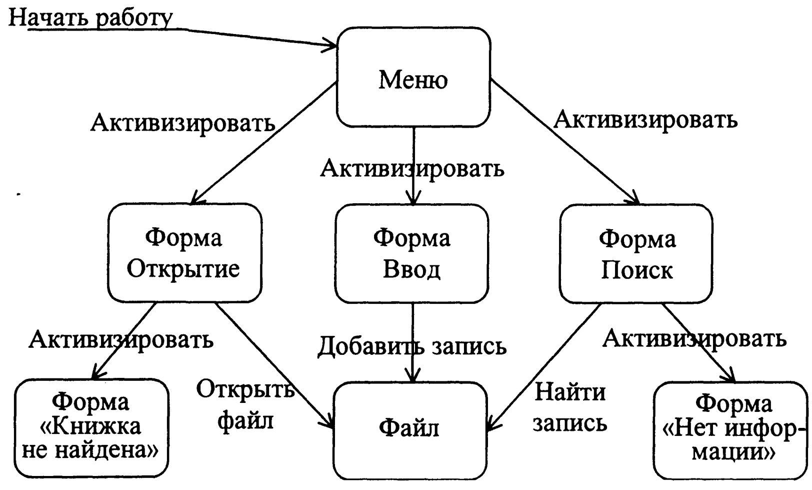


Рис. 1.13. Диаграмма объектов системы «Записная книжка»

3. Процесс декомпозиции прекращается при получении объектов, которые могут быть достаточно просто реализованы, т.е. имеют четко определенную структуру и поведение.

1.4. Объекты и сообщения

В предыдущем разделе было показано, что под объектом применительно к ООП понимается отдельно реализуемая часть предметной области задачи. Разрабатываемая программа, таким образом, состоит из объектов, которые взаимодействуют через передачу сообщений.

Каждый объект, получая сообщения, должен определенным образом «реагировать» на них, выполняя заранее определенные для каждого типа сообщения действия. Например, если объект Форма Открытие из примера 1.5 будет активизирован, то он должен запросить у пользователя имя файла, проверить наличие такого файла и затем либо открыть его, послав соответствующее сообщение объекту Файл, либо активизировать объект Сообщение «Записная книжка не найдена».

Реакция объекта на сообщение может зависеть от его состояния: так объект Файл, получив сообщение «Добавить запись», прежде чем добавлять запись, должен проверить, открыт ли соответствующий файл, и при закрытом файле должен выдать пользователю отказ на выполнение операции добавления.

Состояние объекта характеризуется набором конкретных значений некоторого перечня всех возможных свойств данного объекта, например, состояние колонки (пример 1.3) характеризуется: скоростью обслуживания машин, тем занята колонка или нет, и в занятом состоянии – временем ее освобождения.

Как правило, набор свойств в процессе функционирования не изменяется, могут изменяться лишь конкретные значения этих свойств.

Примечание. Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение, т.е. объект может моделироваться с применением *теории конечных автоматов*.

Поведение объектов, как уже говорилось выше, характеризуется определенным набором реакций на получаемые сообщения и часто зависит от состояния объекта.

Объекты, обладающие сходными состояниями и поведением с точки зрения решаемой задачи, образуют группу (например, объекты Колонки). Свойства объекта, которые отличают его от всех других объектов группы, составляют его *индивидуальность*. Так каждая колонка может иметь собственную пропускную способность.

Если объект может обладать некоторым состоянием, то, соответственно, может возникнуть необходимость в получении информации об этом состоянии. Для получения такой информации объекту посылается сообщение – *запрос*. В

ответ на запрос объект должен переслать отправителю требуемую информацию. В таких случаях говорят, что над объектом выполнена операция *селекции*.

Обращение к объекту для изменения его состояния по всем или отдельным составляющим инициирует выполнение операции *модификации*. Отправитель сообщения-*команды*, реакцией на которую должна быть модификация объекта, может ожидать завершения операции, а может продолжить выполнение своей программы. Второй вариант обработки называется *асинхронным*, и его реализация требует использования (или моделирования) параллельной обработки.

Если объект содержит несколько однотипных компонент, например, массив чисел, то операция, требующая последовательной обработки этих компонент, называется *итерацией*. Поэлементно могут выполняться как операции селекции, так и операции модификации.

Полный список возможных *операций* над объектами выглядит следующим образом (рис. 1.14):

создание объекта;

уничтожение объекта;

модификация – изменение состояния объекта;

селекция – доступ для определения состояния;

итерация – доступ к содержимому объекта по частям в определенной последовательности (используется при наличии в объекте некоторых однотипных компонент).

Соответственно, каждое сообщение, принимаемое объектом, может инициировать выполнение одной или нескольких операций указанных типов.

Например, «Активизировать» для интерфейсного элемента, в зависимости от типа реализации (статической или динамической) может означать, либо создание объекта – при динамической реализации либо его модификацию – «видимый» вместо «невидимый» при статической реализации.

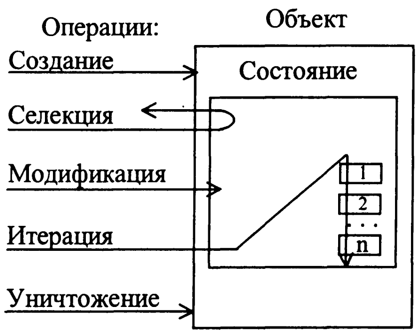


Рис. 1.14. Типы операций над объектом

При выполнении объектной декомпозиции устанавливаются *отношения* между объектами, которые бывают двух типов.

1. В случае, когда один объект передает сообщение другому, говорят, что эти объекты находятся в отношении *использования* (рис.1.15), причем объект, инициирующий сообщение, называется *активным*, а объект, получающий сообщение – *пассивным*.

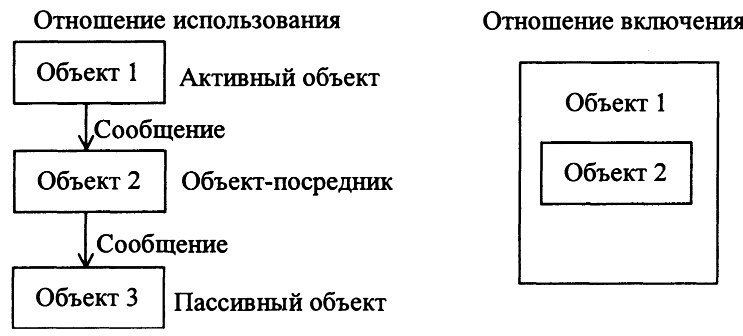


Рис. 1.15. Типы отношений между объектами

Соответственно, отношение использования может принимать форму *воздействия* (активный объект *воздействует* на пассивный объект, передавая ему сообщение), *исполнения* (пассивный объект *исполняет* указание активного объекта) и *посредничества* (некоторый объект – *посредник*, получив сообщение от активного объекта, передает его пассивному объекту).

2. Если объект является результатом декомпозиции более сложного объекта, то говорят, что между этими объектами существует отношение *включения* – первый объект включает второй (иерархия целое/часть).

Виды операций над объектами и типы отношения между ними определяют особенности реализации объектов.

1.5. Классы

Реализация объектов, полученных в результате декомпозиции, принципиально возможна на любом языке, даже на ассемблере. Однако наличие специальных средств позволяет существенно упростить программирование, дополнительно обеспечивая программиста заготовками классов из библиотек и встроенными механизмами обеспечения требуемых свойств.

Классы. Для представления абстракций объектов используется специальный определяемый программистом тип данных – класс.

Класс – это структурный тип данных, который включает описание полей данных, а также процедур и функций, работающих с этими полями данных. Применительно к классам такие процедуры и функции получили название *методов*.

Реализация объединения данных с определенными видами их обработки делает классы пригодными для описания состояния и поведения моделей реальных объектов. Совокупность полей определяется множеством аспектов состояния объекта с точки зрения решаемой задачи, а совокупность методов – множеством аспектов поведения объекта (рис. 1.16).

В программах используются переменные типа класса. Такие переменные принято называть *объектами*.

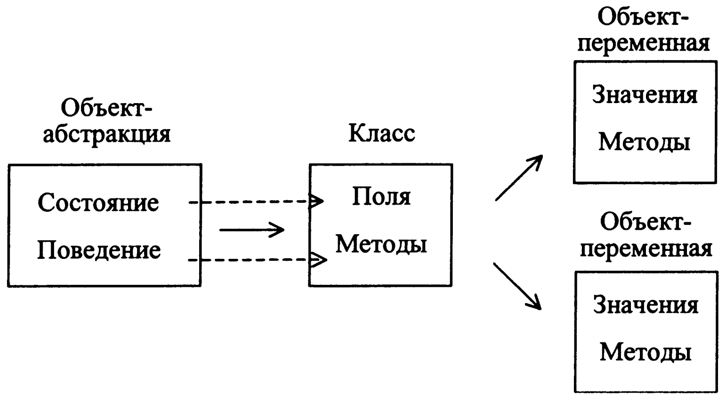


Рис. 1.16. Соответствие объекта-абстракции классу и объектам-переменным

Пример 1.6. Описание класса (класс Окно). Пусть необходимо разработать класс, переменные которого используются для изображения на экране цветного прямоугольника заданного цвета и размера (рис. 1.17).

Проектируемый класс должен содержать поля для сохранения параметров окна: X1, Y1, X2, Y2 – координаты верхнего левого и нижнего правого углов и Color – цвет.

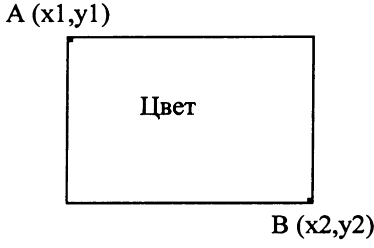


Рис. 1.17. Вид и параметры окна на экране

Пусть единственное сообщение, обрабатываемое объектом, – «Нарисовать окно». Соответственно, класс должен содержать метод, реализующий процесс рисования объекта. Поля объекта можно инициализировать при создании переменной-объекта, передав ему сообщение инициализации, включающее значения полей. Следовательно, класс должен содержать метод инициализации.

Окончательно получаем класс, имеющий следующую структуру:

Класс Окно:
    поля X1,Y1,X2,Y2, Color
    метод Инициализировать(aX1,aY1,aX2,aY2,aColor)
    метод Изобразить
Конец описания.

Создавая объекты типа Окно, инициализируя их в соответствии с условием и посылая им сообщение «Нарисовать окно», получим разные окна на экране, причем параметры этих окон будут храниться в объектах.

Каждая переменная типа класса включает набор полей, объявленных в классе. Совокупность значений, содержащихся в этих полях, моделирует конкретное состояние объекта предметной области. Изменение этих значений в процессе работы отражает изменение состояния моделируемого объекта.

Воздействие на объект выполняется посредством изменения его полей или вызова его методов. Доступ к полям и методам объекта осуществляется, за исключением специальных случаев, с указанием имени объекта (при этом используются составные имена):

<имя объекта>.<имя поля>

или

<имя объекта>.<имя метода>.

Все методы объекта обязательно имеют *доступ ко всем полям своего* объекта. В рассматриваемых далее языках программирования это достигается через *неявную* передачу в метод специального параметра – адреса области данных конкретного объекта (Self – в Паскале и this – в C++). Таким образом, уменьшается количество параметров, явно передаваемых в метод.

Ограничение доступа. Большинство версий объектно-ориентированных языков позволяет ограничить доступ к некоторым полям и методам объекта, обеспечивающим функционирование «внутренностей» объекта. При наличии таких возможностей специальными средствами выделяют *интерфейс* и *реализацию* класса. Описание класса без учета синтаксиса конкретного языка выглядит следующим образом:

Класс <имя класса>
  интерфейс
    <объявление полей и методов класса,
     к которым возможно обращение извне>
  реализация
    <объявление полей и методов класса,
     к которым невозможно обращение извне>
Конец описания.

Как уже говорилось выше, объединение полей данных и процедур и функций, работающих с этими данными, в единый пакет при наличии специальных правил доступа к элементам пакета называется инкапсуляцией.

Наличие интерфейса обеспечивает уменьшение возможности «разрушения» (несанкционированного изменения значений полей) объекта извне. При этом сокрытие особенностей реализации упрощает внесение изменений в реализацию класса, как в процессе отладки, так и при модификации программы. Таким образом, класс определяет существование глобальной области данных внутри объекта, доступной методам объекта.

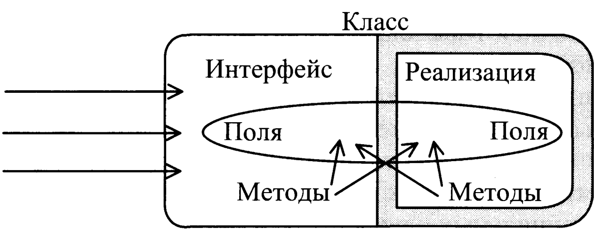


Рис. 1.18. Интерфейс и реализация класса

С другой стороны доступ к объекту регламентируется и должен выполняться через специальный интерфейс (рис. 1.18).

Как и любая переменная программы, объект должен быть размещен в памяти (создан) и удален из памяти (уничтожен). Причем создание и уничтожение объектов выполняется статически или динамически.

Статическое создание объектов выполняется в процессе компиляции программы, а статическое уничтожение – при завершении программы: объект удаляется из памяти вместе с программой.

Динамическое создание и уничтожение объектов выполняется в процессе работы программы специальными командами.

Операция создания и инициализация полей объекта получила название *конструирования* объекта, а операция уничтожения объекта – *деструкции* объекта. Соответствующие методы, если они определены в классе, получили название *конструкторов* и *деструкторов*. Конкретные особенности этих методов в различных языках программирования будут обсуждаться в соответствующих разделах.

Пример 1.7. Сокрытие реализации класса (класс Файл – продолжение примера 1.5). В соответствии с результатами объектной декомпозиции (см. рис. 1.13). Файл должен реагировать на следующие сообщения: «Открыть», «Добавить запись», «Найти запись».

Каждое сообщение должно дополняться необходимой информацией. Так, сообщение «Открыть» должно сопровождаться передачей имени файла. Сообщение «Добавить запись» должно сопровождаться передачей текста записи, состоящей из двух частей: фамилии абонента или названия организации и телефона. Сообщение «Найти запись» должно сопровождаться передачей фамилии абонента или названия организации, по которым должен быть найден телефон. Каждому сообщению должен соответствовать метод, объявленный в интерфейсной части класса и реализующий требуемые действия.

Метод Открыть должен выполнять открытие указанного файла для выполнения операций ввода-вывода. Операция открытия файла требует указания файловой переменной. Поскольку при выполнении операций с этим файлом в дальнейшем должна будет использоваться та же файловая переменная, целесообразно описать ее в секции реализации класса, где она будет доступна методам данного объекта, но не доступна из других объектов программы.

Метод Добавить должен проверять наличие открытого файла и сообщать пользователю о невозможности выполнения операции с закрытым файлом. Если же файл открыт, метод должен выполнить добавление записей в файл. Для выполнения проверки необходимо хранить информацию об открытии файла, для этого можно использовать специальное поле Состояние_файла, которое также можно объявить в секции реализации.

Метод Найти также вначале должен проверять, открыт ли файл. Если файл открыт, метод выполняет поиск информации в файле. Операции проверки открытия файла выполняются в методах Добавить и Найти идентично, их можно выделить в отдельный метод Проверка_открытия, который можно описать в секции реализации.

Для инициализации поля Состояние_файла (в исходном состоянии файл закрыт) можно использовать специальный метод, традиционно с этой целью используется конструктор. При завершении программы файл необходимо закрыть. Операция закрытия может быть описана в деструкторе.

Окончательно для реализации объекта Файл можно предложить класс следующей структуры:

Класс Файл:
 интерфейс
  конструктор Инициализировать;
  метод Открыть (имя файла);
  метод Добавить (фамилия, телефон);
  метод Найти (фамилия);
  деструктор Закрыть_файл;
 реализация
  поле Файловая_переменная;
  поле Состояние_файла;
  метод Проверка_открытия;
Конец описания.

После определения структуры класса должны быть разработаны алгоритмы методов и назначены типы полей и передаваемых параметров.

В программе должна быть использована переменная типа Файл. Так как файл в программе используется постоянно, соответствующую переменную лучше определять статически.

Передача сообщений объекту будет реализована как вызов соответствующего метода интерфейсной части.

1.6. Основные средства разработки классов

Языки, поддерживающие ООП, существенно облегчают разработчику создание новых классов за счет реализации механизмов наследования, композиции, наполнения и полиморфизма.

Наследование. В ООП существует возможность конструирования новых более сложных классов из уже имеющихся посредством добавления полей и определения новых методов (принцип иерархичности). При этом исходный класс, на базе которого выполняется конструирование, часто называется *родителем*, а производный – *потомком*. Специальный механизм наследования обеспечивает классу-потомку возможность использования полей и методов одного или нескольких родительских классов.

Если непосредственный родитель единственный, то наследование называется *простым*, а если таких классов несколько – то *множественным*. При этом класс родитель (или классы родители) и класс потомок образуют иерархию (рис.1.19).

Наличие механизма наследования в языке программирования позволяет повторно не определять в классе уже описанные параметры и свойства объектов, производный класс их просто «наследует».

Пример 1.8. Наследование (класс Окно_меняющее_цвет). Построим на базе класса Окно класс-потомок, который может изменять цвет окна на экране. Для этого к родительскому классу достаточно добавить метод Изменить_цвет:

Класс Окно_меняющее_цвет – родитель: класс Окно:
    метод Изменить_цвет(aColor);
Конец описания.

Класс Окно_меняющее_цвет содержит все поля родительского класса и все его методы. Дополнительно объекты типа Окно_меняющее_цвет могут менять цвет окна на указанный в сообщении «Изменить_цвет» (рис. 1.20).

При множественном наследовании, реализованном, например, в C++, наследуются поля и методы всех родителей. В том случае, если среди родителей есть классы, принадлежащие одной иерархии, происходит дублирование полей и методов, наследуемых от общих родителей. Для того чтобы избежать неоднозначности, в C++ введено понятие *виртуального* наследования, при использовании которого виртуально наследуемые поля и методы не дублируются (более подробное описание и примеры – в разделе 3.3).

Рис. 1.19. Иерархия классов при различных видах наследования

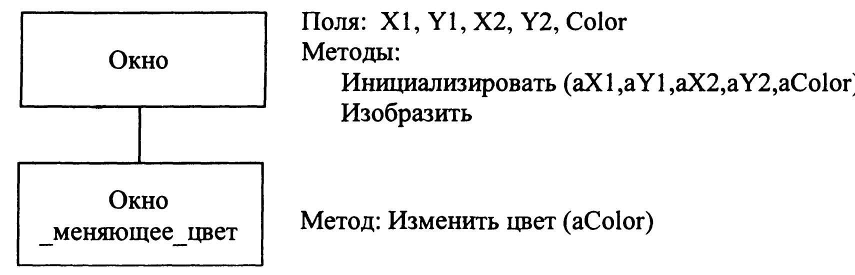


Рис. 1.20. Иерархия классов Окно и Окно_меняющее_цвет

Таким образом, в иерархическом дереве классов по мере удаления от корня мы встречает все более сложные классы, экземплярами которых будут объекты с более сложной структурой и поведением.

Наследование свойств в иерархии существенно упрощает работу программиста. В настоящее время созданы библиотеки наиболее часто встречающихся классов, которые можно использовать вновь и вновь, строя на их основе классы для решения различных задач.

Простой полиморфизм. При создании иерархии классов может обнаружиться, что некоторые свойства объектов, сохраняя название, изменяются по сути.

Для реализации таких иерархий в языке программирования должен быть предусмотрен *полиморфизм,* обеспечивающий возможность задания различных реализаций некоторого единого по названию метода для классов различных уровней иерархии. В ООП такой полиморфизм называется *простым,* а методы, имеющие одинаковое название – *статическими полиморфными.* Совокупность полиморфных методов с одним именем для иерархии классов образует единый полиморфный метод иерархии, в котором реализация полиморфного метода для конкретного класса представляет отдельный *аспект.*

Примечание. Термин «полиморфизм» в программировании, в соответствии со своим изначальным смыслом («многообразие»), используется для обозначения встроенного механизма определения соответствия кода функции типу параметров. Такой механизм реализуется не только в средствах ООП.

Различают несколько терминов, связанных с конкретными механизмами реализации полиморфизма для различных случаев:

чистый полиморфизм – используется для обозначения того, что один код функции может по-разному интерпретироваться в зависимости от типа аргументов; используется в языках высокого уровня абстракции, например, в языке LISP или SMALLTALK;

перегрузка (*полиморфные имена функций*) – используется, когда определяется несколько функций с одним именем – одно и то же имя функции может многократно использоваться в разных местах программы; выбор нужной функции может определять типами аргументов, областью видимости (внутри модуля, файла, класса и т.д.); если выбор определяется типом

аргументов, то перегрузка называется *параметрической*; например, язык C++ позволяет разработчику выполнять параметрическую перегрузку функций вне классов;

переопределение (простой полиформизм) – используется в ООП при наличии различных определений методов в иерархии классов, конкретный метод определяется типом объекта при компиляции программы (*раннее связывание*), методы называются *статическими полиморфными*;

полиморфные объекты (*сложный полиморфизм)* – используются в ООП при наличии различных определений методов в иерархии классов – конкретный метод также определяется типом объекта, но в процессе выполнения программы (*позднее связывание*),методы называются *виртуальными полиморфными* (рассмотрены далее);

обобщенные функции или *шаблоны* – используются в ООП при реализации в языке параметризованных классов (например, в C++), параметрами такого класса являются типы аргументов методов класса (рассмотрены далее).

Пример 1.9. Простой полиморфизм (класс Окно_с_текстом). Пусть необходимо разработать на базе класса Окно класс Окно_с_текстом. Для этого к полям класса Окно необходимо добавить специальные поля для определения координат первой буквы текста – Xt, Yt и поле, содержащее сам текст – Text. Кроме этого, понадобится специальный метод, который будет обрабатывать сообщение «Нарисовать». Однако у нас уже существует родительский метод Изобразить (!), который обрабатывает это сообщение. Следовательно, необходимо заменить родительский метод методом потомка. Механизм полиморфизма позволяет для класса-потомка Окно_с_текстом предусмотреть собственный метод Изобразить (рис. 1.21).

Метод Инициализировать также должен быть переопределен, так как он должен инициализировать дополнительные поля класса.

Класс Окно_с_текстом – родитель: класс Окно:
  поля Xt, Yt, Text
  метод Инициализировать (aX1,aY1,aX2,aY2,aColor,aXt, aYt, aText)
  метод Изобразить
Конец описания.

Примечание. При реализации методов Изобразить и Инициализировать потомка можно вызвать соответствующие родительские методы, а затем добавить операторы, определяющие собственные действия метода для разрабатываемого класса.

Сложный полиморфизм или создание полиморфных объектов. *Полиморфными объектами* или *полиморфными переменными* называются переменные, которым в процессе выполнения программы может быть присвоено значение, тип которого отличается от типа переменной. В языках с жесткой типизацией такая ситуация может возникнуть:

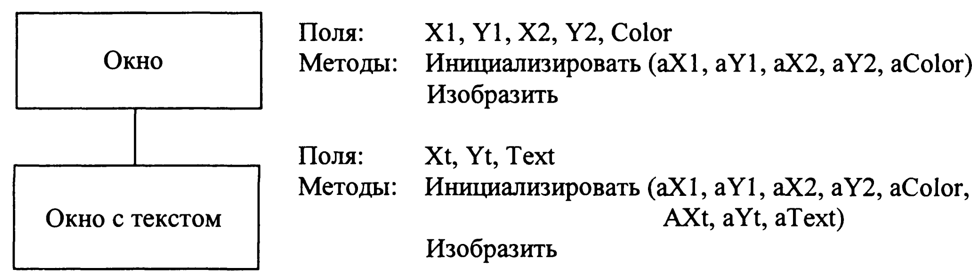


Рис. 1.21. Иерархия классов Окно и Окно_с_текстом

при передаче объекта типа класса-потомка в качестве фактического параметра подпрограмме, в которой этот параметр описан как параметр типа класса-родителя (явно – в списке параметров или неявно – в качестве внутреннего параметра, используемого при вызове методов – self или this);

при работе с указателями, когда указателю на объект класса-родителя присваивается адрес объекта класса-потомка.

Примечание. В принципе, в обоих случаях речь идет об одной и той же ситуации: во внутреннем представлении в любом из указанных случаев используются адреса объектов (подробности – в разделах 2.3, 3.4).

Тип полиморфного объекта становится известным только на этапе выполнения программы. Соответственно, при вызове полиморфного метода для такого объекта нужный аспект также должен определяться на этапе выполнения. Для этого в языке должен быть реализован механизм *позднего связывания*, позволяющий определять тип объекта и, соответственно, аспект полиморфного метода, к которому идет обращение в программе, *на этапе ее выполнения*.

С помощью механизма позднего связывания реализуется оперативная перестройка программы в соответствии с типами используемых объектов. Поясним сказанное на примере.

Пример 1.10. Сложный полиморфизм. Пусть родительский класс содержит два метода Out и Print, один из которых вызывает другой. Класс-потомок наследует метод Out, но имеет собственный метод Print. Метод Out наследуется и может быть вызван как для объекта класса-родителя, так и для объекта класса-потомка (рис. 1.22).

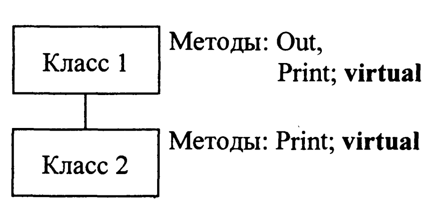


Рис. 1.22. Иерархия классов примера 1.10

При вызове метода Out для объекта класса-потомка необходимо обеспечить, чтобы этот метод вызывал метод Print потомка, а не родителя (рис. 1.23).

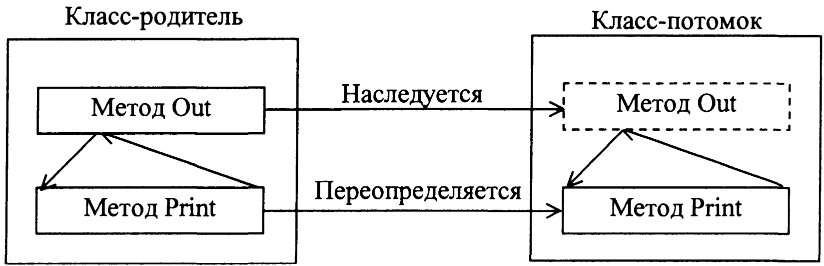


Рис. 1.23. Пример необходимости позднего связывания

Определить, для объекта какого класса: родителя или потомка вызывается метод Out, можно только на этапе выполнения. Следовательно, для метода Print необходимо обеспечить позднее связывание.

Если определение адреса метода Print происходило бы на этапе компиляции программы, то и для объекта родительского класса, и для объекта класса-потомка из метода Out вызывался бы метод Print класса-родителя. Описание метода Print как метода, для которого запрещается определение адреса на этапе компиляции, приведет к тому, что адрес метода Print будет определяться в процессе выполнения программы. В этот момент уже будет известно, объект какого класса вызывает метод Out, и будет вызываться метод Print именно этого класса.

Методы, для которых должно реализовываться позднее связывание, получили название *виртуальных*. Для их описания в рассматриваемых далее языках программирования используется служебное слово **virtual**.

Следует отметить, что методы, работающие с полиморфными объектами – это всегда методы классов-родителей, описывающие общие моменты поведения объектов. В сложной иерархии, таким образом, можно выделить *семейство классов* со схожим поведением объектов. Они образуют поддерево, в корне которого находится класс, определяющий общие моменты поведения.

Реализация механизма позднего связывания осуществляется с использованием специальной таблицы, получившей название *таблицы виртуальных методов* (ТВМ). ТВМ создается для каждого класса, имеющего собственные или наследующего виртуальные методы. Она содержит адреса виртуальных методов (рис. 1.24). Объекты таких классов содержат адрес ТВМ своего класса. При вызове виртуального метода для объекта происходит обращение к ТВМ класса, по которой и определяется требуемый метод.

При использовании полиморфных объектов возникают проблемы с доступом к полям объекта, описанным в классе-потомке: указатель на объект класса-родителя связан с описанием полей класса-родителя, и поля, описанные в классе-потомке, для него «невидимы» (рис. 1.25).

В таких случаях приходится средствами используемого языка *явно переопределять тип* объекта.

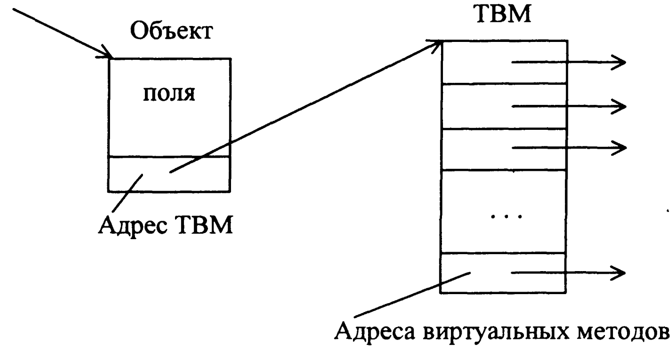


Рис. 1.24. Реализация механизма позднего связывания

Композиция. В разделе 1.3 указано, что в результате объектной декомпозиции второго и более уровней могут быть получены объекты, находящиеся между собой в отношении включения (см. рис. 1.15). Классы для реализации таких объектов могут строиться двумя способами: с использованием наследования или композиции.

Применение наследования эффективно в том случае, если разрабатываемый класс имеет с исходным сходную структуру и элементы поведения, например, Окно и Окно_меняющее_цвет.

В тех случаях, когда сходное поведения не просматривается или наследование по каким-либо причинам нецелесообразно, можно использовать композицию классов.

Композицией называется такое отношение между классами, когда один является частью второго. Конкретно, композиция реализуется включением в класс поля, являющегося объектом другого класса. Такие поля обычно называют *объектными полями*.

Указатель на объект класса-родителя
Шаблон указателя на объект класса-родителя
«Невидимые» поля
Объект класса-потомка
Поля, наследуемые от родителя
Собственные поля потомка

Рис. 1.25. Необходимость явного указания типа объекта, адресуемого указателем родительского класса

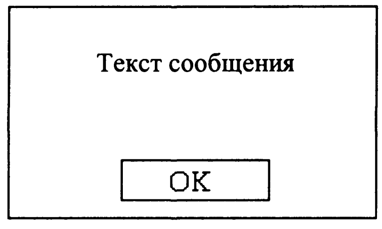


Рис. 1.26. Вид окна сообщения

Пример 1.11. Композиция (класс Сообщение – продолжение примера 1.5). Визуально сообщение обычно выглядит, как окно с текстом и кнопкой подтверждения (рис. 1.26).

При разработке класса Сообщения попытаемся использовать уже описанный класс Окно_с_текстом. Окно сообщения без кнопки представляет собой объект класса Окно_с_текстом. Кнопка также представляет собой Окно_с_текстом. Попытка использования при разработке множественного наследования приведет к дублированию полей. Чтобы этого избежать, используем для хранения параметров изображения кнопки поле типа Окно_с_текстом.

Класс Сообщение – родитель: класс Окно_с_текстом:
  поле Кнопка: **Окно_с_текстом**
  метод Инициализировать
      (aX1,aY1,aX2,aY2,aColor,aXt,aYt,aText,
      bX1,bY1,bX2,bY2,bColor,bXt,bYt,bText)
  метод Изобразить
Конец описания.

Метод Инициализировать при этом должен получить двойной список параметров по сравнению с методом Инициализировать класса родителя. Первым набором инициализируются родительские поля, а вторым – аналогичные поля включенного объекта (рис. 1.27).

Метод Изобразить должен выводить на экран оба окна: одно – используя родительский метод, второе – используя метод поля-объекта.

Объект класса Сообщение

	Кнопка
X1	X1
Y1	Y1
X2	X2
Y2	Y2
Color	Color
Xt	Xt
Yt	Yt
Text	Text

Рис. 1.27. Структура полей класса Сообщение

Доступ к компонентам объектного поля осуществляется с указанием имени объекта, имени поля, имени компонента:

<имя объекта>.<имя поля>.<имя компонента>

или

<имя объекта>.<имя поля>.<имя метода>.

Возможно произвольное количество вложений объектных полей.

Наполнение. Включение объектов в некоторый класс можно реализовать и с

использованием указателей на эти объекты. В отличие от объектного поля, которое включает в класс точно указанное количество объектов (1 или более – при использовании массива объектов) конкретного класса, использование указателей позволяет включить 0 или более объектов, если они собраны в массив или списковую (линейную или нелинейную) структуру.

Пример 1.12. Наполнение (класс Функция). В качестве примера рассмотрим класс, объекты которого должны осуществлять разбор заданного алгебраического выражения (выражение представляет собой запись функции от одной переменной – x). Данный объект должен обрабатывать сообщения:

1. «Конструировать» – в процессе конструирования должно строиться бинарное дерево вычисления выражения (рис. 1.28), переданного в списке параметров.

2. «Вычислить» – при обработке этого сообщения должно вычисляться значение функции по заданному значению аргумента.

В качестве элементов дерева будем использовать объекты класса Триада. Объявление класса Триада, реализующего элемент дерева вычислений, может выглядеть следующим образом:

Класс Триада:
  поле-адрес Левое_поддерево: **Триада**
  поле-адрес Правое_Поддерево: **Триада**
  поле Операция
  поле Результат
  метод Инициализировать (выражение)
Конец описания.

При описании класса Функция также используется поле-адрес:

Класс Функция:
  поле-адрес Корень_дерева: **ТРИАДА;**
  метод Конструировать (выражение) ;
  метод Вычислить;
Конец описания.

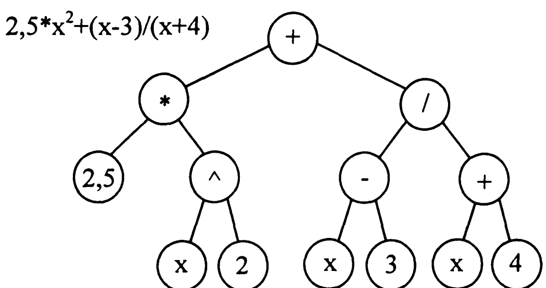


Рис. 1.28. Дерево вычисления выражения

Механизм наполнения в основном используется для подключения объекта или структуры объектов к некоторому классу, реализующему управление сразу всей структурой.

1.7. Дополнительные средства и приемы разработки классов

Рассмотренные в предыдущем разделе средства разработки классов являются основными. Они предусматриваются принципами ООП. Однако к настоящему моменту постепенно складывается более сложная модель ООП, включающая такие понятия, как *метаклассы*, *контейнерные классы, делегирование методов, параметризованные классы.* Появление в языках программирования соответствующих средств позволяет создавать более эффективные программы.

Метаклассы. Дальнейшее развитие идеи реализации полиморфных объектов привело к появлению более высокого уровня абстракции – метаклассов. *Метакласс* – тип, значениями которого являются классы, как ссылки на типы. Переменные типа метакласса можно использовать вместо явного указания класса в соответствии с традиционными правилами *косвенного* доступа (рис. 1.29).

Реализация метаклассов базируется на использовании специальных таблиц, в которых хранится информация о классе: имя класса, имя класса-родителя, адреса методов и т.д. Поскольку эти таблицы используются во время выполнения программы, то они получили название RTTI (Run Time Type Information – «информация о типе времени выполнения»).

Эти же таблицы используются для реализации следующих операций:

операция проверки принадлежности объекта заданному классу или его потомкам;

операция уточнения класса объекта – отличается от операции явного переопределения типа тем, что перед переопределением проверяется, принадлежит ли объект данному классу или его потомкам;

специальные операции определения, модификации или проверки свойств класса (как типа) – для реализации этих операций язык программирования

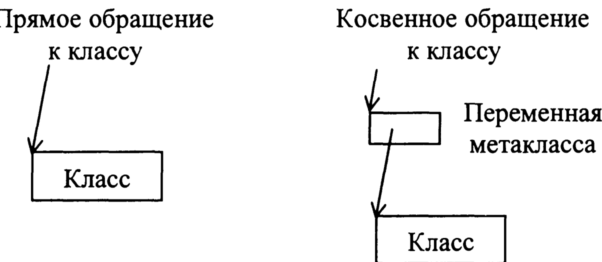


Рис. 1.29. Организация прямого и косвенного доступа к классу

должен предоставлять возможность описания *полей* и *методов класса*, обращение к которым возможно при отсутствии объектов класса.

Делегирование методов. *Делегирование* – способ заимствования методов у объектов других классов. Оно представляет собой альтернативу переопределению методов, используемому полиморфными объектами. В отличие от переопределения, делегирование позволяет определять *различное поведение объектов, принадлежащих одному классу.* Причем заимствование методов возможно как в пределах класса или иерархии классов, так и у объектов классов других иерархий.

Метод при этом вызывается косвенно, через указатель на него. Язык, в котором возможна реализация делегирования, должен обеспечивать возможность определения *указателей на методы*. Назначение или замена метода осуществляются присваиванием соответственного значения специальному указателю (рис. 1.30).

Следует различать *статическое* и *динамическое* делегирование. При статическом делегировании соответствующий указатель инициализируется в процессе компиляции программы и при выполнении программы не меняется. При динамическом делегировании значение указателю присваивается в процессе выполнения программы и может изменяться в зависимости от ситуации.

Статическое делегирование используется в двух случаях:

1) если требуемое поведение объекта уже описывалось для объектов другого класса – в этом случае делегирование позволяет избавиться от повторов в программе;

Примечание. В С++ в аналогичных случаях возможно использование множественного наследования, но оно может оказаться неэффективным, поскольку вместе с желаемым поведением придется унаследовать и все остальные элементы класса.

2) если класс объявлен с не полностью определенным поведением объек-

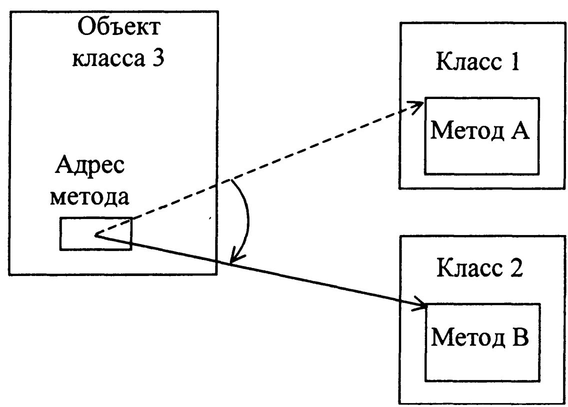


Рис. 1.30. Делегирование метода

тов (обычно так описываются некоторые библиотечные классы) и его поведение уточняется для конкретных экземпляров объектов.

Динамическое делегирование используется при создании объектов с изменяемым поведением, когда конкретный вариант поведения определяется некоторыми внешними обстоятельствами.

Пример 1.13. Делегирование методов (класс Фигура). В примере 1.3 рассматривалась объектная декомпозиция простейшего текстового редактора, позволяющего рисовать окружности и квадраты с изменяющимися размером, цветом и положением. В результате этой декомпозиции были получены три объекта: Монитор, Круг и Квадрат (см. рис. 1.10). Те же действия будет выполнять программа, при разработке которой использован объект с динамическим поведением. Объектная декомпозиция текстового редактора в этом случае будет включать два объекта – Монитор и Фигура (рис. 1.31).

Вариант 1. Класс Фигура описан без указания конкретного варианта выводимой фигуры.

Класс Фигура1:
 реализация
  поля X, Y, R, Color
  поле Адрес_метода_рисования
 интерфейс
  конструктор Создать (aX, aY, aR, aColor,
                  аАдрес_метода_рисования)
  метод Изменить_цвет (aColor)
  метод Изменить_размер (aR)
  метод Изменить_местоположение (aX, aY)
Конец описания.

Адрес конкретного метода рисования должен указываться при создании соответствующего объекта. Изменение этого адреса в процессе выполнения программы не предусмотрено. В данном случае используется статическое делегирование.

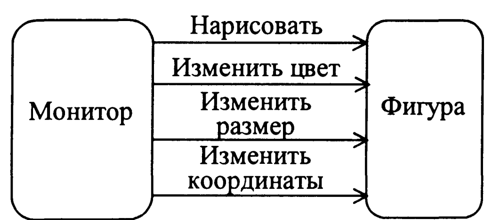


Рис. 1.31. Объектная декомпозиция графического редактора

Вариант 2. Возможен другой вариант: класс Фигура содержит метод, который меняет используемую процедуру рисования.

```
Класс Фигура2 :
 реализация
   поля x, y, r, Color
   поле Адрес_метода_рисования
 интерфейс
   конструктор Создать (ax, ay, ar, aColor,
                              aАдрес_метода_рисования)
   метод Изменить_цвет (aColor)
   метод Изменить_размер (aR)
   метод Изменить_местоположение (aX, aY)
   метод Изменить_тип_Фигуры (aАдрес_метода_рисования)
Конец описания.
```

Выполнив указанный метод для объектов данного класса, можно изменить процедуру рисования в процессе выполнения программы – динамическое делегирование.

Сами методы рисования могут быть определены как в том же, так и в другом классе, например:

```
Класс Методы_рисования:
  интерфейс
    метод Рисование_окружности(aX, aY, aR, aColor)
    метод Рисование_квадрата(aX, aY, aR, aColor)
Конец_описания.
```

При определении метода в другом классе, к сожалению, приходится в явном виде передавать все необходимые параметры рисования, так как в противном случае последние будут недоступны. При определении делегируемых методов в своем классе этой проблемы не возникает.

Вариант 3.

```
Класс Фигура3:
 реализация
   поля X, Y, R, Color
   поле Адрес_метода_рисования
   метод Рисование_окружности
   метод Рисование_квадрата
 интерфейс
   конструктор Создать(aX, aY, aR, aColor, aFigura_type)
   метод Изменить_цвет (aColor)
```

метод Изменить_размер (aR)
метод Изменить_местоположение (aX, aY)
метод Изменить_тип_фигуры (aТип_фигуры)

Конец описания.

Параметр aFigura_type будет получать значение, определяющее подключаемый к объекту метод рисования.

При использовании делегирования необходимо следить, чтобы заголовок метода соответствовал типу указателя, используемого для хранения адреса метода.

Контейнерные классы. *Контейнеры* – это специальным образом организованные объекты, используемые для хранения объектов других классов. Для реализации контейнеров разрабатываются специальные контейнерные классы. Контейнерный класс обычно включает набор методов, позволяющих выполнять некоторые операции, как с отдельным объектом, так и с группой объектов. В виде контейнеров, как правило, реализуются сложные структуры данных (различные виды списков, динамических массивов и т.п.). Разработчик наследует от класса-элемента класс, в который добавляются нужные ему информационные поля, и получает требуемую структуру. При необходимости он может наследовать класс и от контейнерного класса, добавляя к нему свои методы (рис. 1.32).

Контейнерный класс обычно включает методы создания, добавления и удаления элементов. Кроме того, он должен обеспечивать поэлементную обработку (например, поиск, сортировку). Все методы программируются через указатели на класс-элемент. Методы добавления и удаления элементов при выполнении операций обычно обращаются к специальным полям класса-элемента, используемым для создания структуры (например, для односвязного списка – к полю, хранящему адрес следующего элемента).

Методы, реализующие поэлементную обработку, должны работать с полями данных, определенными в классах-потомках класса-элемента.

Поэлементную обработку реализуемой структуры можно реализовать двумя способами. Первый способ – универсальный – заключается в использовании *итераторов,* второй – в определении специального метода, который содержит в списке параметров адрес процедуры обработки.

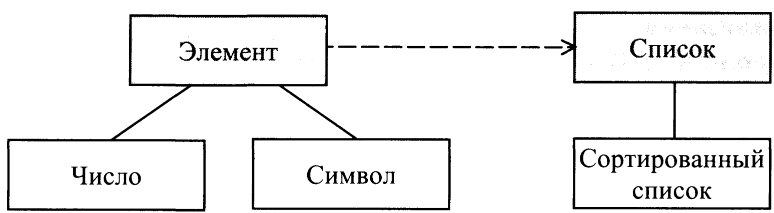


Рис. 1.32. Построение классов на базе контейнерного класса и класса элемента

Теоретически итератор должен обеспечивать возможность реализации циклических действий следующего вида:

<очередной элемент>:=<первый элемент>
цикл-пока <очередной элемент> определен
<выполнить обработку>
<очередной элемент>:=<следующий элемент>
все-цикл.

Поэтому обычно он состоит из трех частей: метод, позволяющий организовать обработку данных с начала (получение первого элемента структуры); метод, организующий переход к следующему элементу, и метод, позволяющий проверить окончание данных. Доступ к очередной порции данных при этом осуществляется через специальный указатель текущей порции данных (указатель на объект класса-элемента).

Пример 1.14. Контейнерный класс с итератором (класс Список). Разработаем контейнерный класс Список, реализующий линейный односвязный список из объектов класса Элемент.

Класс Элемент:
поле Указатель_на_следующий
Конец описания.

Класс Список включает три метода, составляющих итератор: метод Определить_первый, который должен возвращать указатель на первый элемент, метод Определить_следующий, который должен возвращать указатель на следующий элемент, и метод Конец_списка, который должен возвращать «да», если список исчерпан.

Класс Список
реализация
поля Указатель_на_первый, Указатель_на_текущий
интерфейс
метод Добавить_перед_первым (аЭлемент)
метод Удалить_последний
метод Определить_первый
метод Определить_следующий
метод Конец_списка
Конец описания.

Тогда поэлементная обработка списка будет программироваться следующим образом:

Элемент:= Определить_первый
цикл-пока не Конец_списка
Обработать элемент, переопределив его тип

Элемент:= Определить_следующий
все-цикл

При использовании второго способа поэлементной обработки реализуемой структуры процедура обработки элемента передается в списке параметров. Такую процедуру можно определить, если известен тип обработки, например, процедура вывода значений информационных полей объекта. Процедура должна вызываться из метода для каждого элемента данных. В языках с жесткой типизацией данных тип процедуры должен описываться заранее, при этом часто невозможно предусмотреть, какие дополнительные параметры должны передаваться в процедуру. В таких случаях первый способ может оказаться предпочтительнее.

Пример 1.15. Контейнерный класс с процедурой обработки всех объектов (класс Список). В этом случае класс Список будет описываться следующим образом:

Класс Список
реализация
поля Указатель_на_первый, Указатель_на_текущий
интерфейс
метод Добавить_перед_первым(аЭлемент)
метод Удалить_последний
метод Выполнить_для_всех(аПроцедура_обработки)
Конец описания.

Соответственно, тип процедуры обработки должен быть описан заранее, с учетом того, что она должна получать через параметр адрес обрабатываемого элемента, например:

Процедура_обработки (аЭлемент)

Использование полиморфных объектов при создании контейнеров позволяет создавать достаточно универсальные классы.

Параметризованные классы. *Параметризованный класс* (или *шаблон*) представляет собой определение класса, в котором часть используемых типов компонент класса определяется через параметры. Таким образом, каждый шаблон определяет группу классов, которые, несмотря на различие типов, характеризуются одинаковым поведением. Переопределить тип в процессе выполнения программы нельзя: все операции конкретизации типа выполняются компилятором (точнее – препроцессором) C++.

Параметризованные классы реализованы в C++. Они часто используются для реализации контейнерных классов, причем обычно в качестве элемента выступает полиморфный объект, указатель на базовый класс которого и

передается через параметр. Классы, полученные из такого шаблона, могут оперировать как с объектами базового класса, так и с объектами производных классов.

Пример 1.16. Шаблон классов (шаблон классов Список). Шаблон практически полностью повторяет описание класса, но везде, где должен указываться тип элемента, вместо него следует указать параметр (в нашем случае **Тип_элемента**):

Шаблон классов Список (Тип_элемента)
реализация
поле Указатель_на_первый: Указатель на **Тип_элемента**
поле Указатель_на_текущий: Указатель на **Тип_элемента**
интерфейс
метод Добавить_перед_первым
(аЭлемент: Указатель на **Тип_элемента**)
метод Удалить_последний: Указатель на **Тип_элемента**
метод Определить_первый: Указатель на **Тип_элемента**
метод Определить_следующий: Указатель на **Тип_элемента**
метод Конец_списка
Конец описания.

При использовании шаблона указывается его имя и соответствующее значение параметра, например:

Список (Запись1)

или

Список (Запись2)

Реализация контейнеров в виде параметризованных классов по сравнению с обычной реализацией может оказаться более наглядной и, следовательно, предпочтительной.

Исключения. При выполнении любой сложной программы возможно возникновение ситуаций, нарушающих нормальный процесс обработки (например, отсутствует файл данных, делитель выражения равен нулю и т.п.). Такие ситуации принято называть *исключительными*.

Правильно написанная программа должна предусматривать корректирующие действия в подобных случаях, например, выдавать соответствующую информацию пользователю и предлагать ему варианты выхода из аварийной ситуации.

В небольших программах такие действия предусмотреть несложно. В больших же программах, в том числе и программах, написанных с

использованием ООП, обработка аварийных ситуаций превращается в достаточно сложную задачу, так как нарушения обнаруживаются в одном месте программы (объектом, выполняющим обработку), а возможная коррекция должна быть предусмотрена в другом (в объекте, инициирующем данную обработку).

Для программирования корректирующих действий в таких случаях используют *механизм исключений*. Этот механизм базируется на том, что обработка некорректных ситуаций выполняется в два этапа:

генерация информации о возникновении исключительной ситуации – *генерация исключения*;

обработка этой информации – *перехват исключения*.

При перехвате исключений используется стек вызовов, в котором в момент передачи управления очередной подпрограмме фиксируется адрес возврата. Например, если подпрограмма (или метод) А вызывает подпрограмму (или метод) В, а та в свою очередь вызывает подпрограмму (или метод) С, то в стеке последовательно записываются: адрес возврата в А и адрес возврата в В (рис. 1.33).

Ошибка, обнаруженная в подпрограмме С, может быть обработана в самой подпрограмме С, а может быть передана для обработки подпрограммой В или подпрограммой А, или даже подпрограммами, вызвавшими А. Для этого в одной из этих подпрограмм необходимо предусмотреть обработчик исключительных ситуаций, соответствующий по типу обнаруженной ошибке.

Поиск нужного обработчика осуществляется следующим образом. Вначале проверяется, не описан ли требуемый обработчик в подпрограмме С. Если обработчик отсутствует, то по стеку вызовов определяется подпрограмма, вызвавшая подпрограмму С, и проверяется наличие обработчика в ней. Не обнаружив обработчика в подпрограмме В, система по стеку определит подпрограмму, вызвавшую В, и продолжит поиск обработчика в ней. Обратный просмотр стека будет продолжаться до тех пор, пока не обнаружится обработчик требуемого типа или не выяснится, что такой обработчик в программе не определен. В последнем случае программа завершается аварийно.

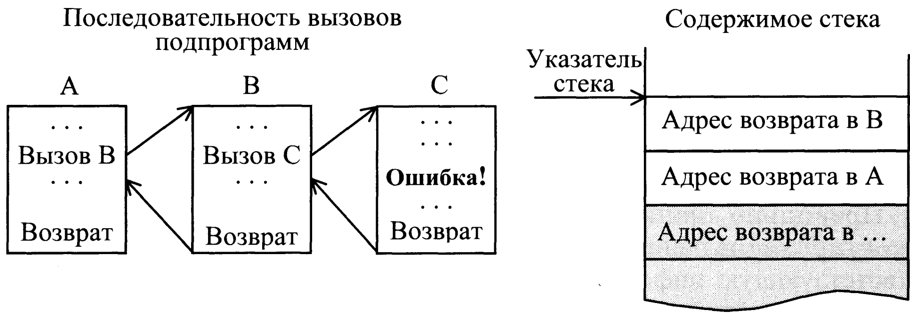


Рис. 1.33. Организация стека вызовов

Механизм исключений предполагает использование двух конструкций. Первая – обеспечивает описанный выше алгоритм поиска обработчика исключения. Она может быть названа *обрабатывающей* конструкцией (**try...except** – в Delphi Pascal, **try... catch** – в стандарте C++, **__try ... __ except** – в стандарте C). Операторы обработки исключения выполняются только, если при выполнении заданного фрагмента программы обнаруживается исключение указанного типа.

Вторая – используется в тех случаях, когда необходимо обеспечить правильное освобождение ресурсов метода даже при обнаружении исключительных ситуаций. Она называется *завершающей* конструкцией обработки исключения (**try... finally** – в Delphi Pascal, **__try...__finally** – в стандарте C, **try...__finally** – в стандарте C++Builder). Завершающая обработка исключения отличается от обычной тем, что блок операторов завершающей обработки выполняется в любом случае: как при обнаружении исключения, так и при его отсутствии. Эта конструкция обычно используется для обеспечения освобождения памяти, закрытия файлов и т.д.

В обоих случаях в месте обнаружения исключительной ситуации программист организует *генерацию исключения* (**raise** – в Delphi Pascal, **throw** – в C++, вызывает функцию **RaiseException** – в C). При генерации исключения создается некоторый объект, содержащий информацию об обнаруженной ситуации. В простейшем случае таким объектом может служить скалярная переменная одного из стандартных типов, а в более сложных – объект описанного ранее класса. В качестве информации может использоваться номер исключения, строка сообщения, значения операндов невыполненной операции, адрес некорректных данных и т.д.

Затем осуществляется обратный просмотр стека вызовов подпрограмм вплоть до основной программы и поиск ближайшего фрагмента, предусматривающего действия по обработке исключений. При этом возможны три варианта:

фрагмент, предусматривающий обработку исключений требуемого типа не обнаружен – выполняется аварийное завершение программы с выдачей предусмотренной по умолчанию информации;

обнаружен фрагмент, включающий обрабатывающую конструкцию – исключение корректируется, и выполнение программы продолжается;

обнаружен фрагмент, включающий завершающую конструкцию – выполняются операторы завершающей обработки и программа продолжается с операторов, следующих за блоком завершения.

Большое внимание при программировании обработки исключений уделяется работе с типами исключений. Наибольший интерес представляет объявление иерархии классов исключений, что позволяет перехватывать сразу все исключения типов иерархии обработчиком типа указатель (ссылка) на базовый класс. В процессе обработки таких иерархий исключений обычно

используют динамические полиморфные методы, вызов которых происходит в соответствии с реальным типом (классом) исключения, определяемым на этапе выполнения программы.

Вопросы для самоконтроля

1. Определите процедурную и объектную декомпозицию предметной области задачи. Чем они различаются? Назовите достоинства и недостатки этих способов декомпозиции.

2. Назовите семь основных принципов ООП и прокомментируйте, как они использованы.

3. Что такое объект и каким образом объекты соединяются в систему для решения задачи? Чем характеризуется объект?

4. Определите понятие «класс». Чем классы отличаются от других типов данных?

5. Как связаны между собой объект предметной области, класс и программный объект? Каким образом в программных объектах реализуются состояние, поведение и идентификация объектов предметной области? Назовите операции, которые могут быть выполнены над программными объектами.

6. Определите основные средства разработки классов. Почему они названы основными? Охарактеризуйте каждое из перечисленных средств и поясните в каких ситуациях их целесообразно использовать.

7. Какие дополнительные средства разработки классов появились в последние годы? Для чего они могут быть использованы?

8. Назовите основные этапы разработки программных систем с использованием ООП и расскажите о каждом из них.

2. СРЕДСТВА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ В BORLAND PASCAL 7.0

При разработке языка Borland Pascal 7.0 использована логически завершенная, но самая простая объектная модель. В этой модели отсутствуют специальные средства сокрытия реализации классов и поддержки дополнительных средств и приемов разработки классов (абстракций метаклассов, делегирования и т. д.).

Задача данной главы показать преимущества даже такой упрощенной модели ООП по сравнению с ранее известными подходами в программировании.

2.1. Определение класса

Итак, класс в ООП – это структурный тип данных, который включает описание полей данных, процедур и функций, работающих с этими полями данных.

Определение класса в Borland Pascal 7.0 осуществляется в два этапа. На первом этапе описывается структура класса, где указываются: имя класса, поля данных и *прототипы* (заголовки) методов:

```
Type
  <имя класса> = object
      <имя поля данных 1> : <тип данных 1>;
                        . . .
      <имя поля данных N> : < тип данных N>;
      Procedure <имя метода 1>(<список параметров>);
      Function <имя метода 2>(<список параметров>):<тип функции>;
                        . . .
      Procedure <имя метода L>(<список параметров>);
                        . . .
      Function <имя метода M>(<список параметров>):<тип функции>;
End;
```

Поля данных класса могут быть следующих типов: числового (*byte, shortint, word, integer, longInt, real, single, double, extended, comp*); логического (*boolean*); символьного (*char*); строкового (*string*); адресного (*pointer*); типа диапазон; множество; массив; файл (*text, file, file of* ...); класс; указатель на класс и др.

Методами являются процедуры и функции языка Borland Pascal 7.0. При описании структуры класса указываются только заголовки методов (имя процедуры или функции, список передаваемых параметров, тип функции).

На втором этапе описываются тела методов, причем перед именем метода через точку указывается имя класса, а список параметров и тип функции можно опустить.

Procedure <имя класса>.<имя метода>;
<описание локальных переменных, процедур и функций>
Begin <операторы> **End**;
или
Function <имя класса>.<имя метода>;
<описание локальных переменных, процедур и функций>
Begin <операторы> **End**;

Данные и методы описываются в одном классе. Такое объявление объединяет данные с процедурами и функциями, манипулирующими этими данными. При этом все данные, описанные внутри класса, автоматически становятся глобальными по отношению к его методам, т.е. общедоступными как для процедур, так и для функций класса.

Описав новый тип – класс, мы получаем возможность объявлять переменные этого класса. Переменную типа класса принято называть экземпляром класса или *объектом*. Объекты можно создавать как в статической, так и в динамической памяти. При необходимости можно создавать массивы объектов, строить из них списки, включать их или указатели на них в качестве полей в записи или другие классы.

Обращение к полям и методам класса осуществляется с указанием имени переменной (объекта), по аналогии с обращением к полям записи с использованием:

– составных имен: <имя объекта>.<имя поля>

или

<имя объекта>.<имя метода>(<список фактических параметров>);

– оператора присоединения with:

with <имя объекта> **do**
begin
... <имя поля> ...
... <имя метода> (<список фактических параметров>) ...
end;

Для инициализации полей объектов можно использовать три способа:

– инициализировать поля в основной программе операторами присваивания вида:

<имя объекта>.<имя поля> := <значение>

– разработать специальный метод инициализации, который получает список параметров и инициализирует ими поля (такой метод обычно называют Init);

– использовать типизированные константы вида:

Const <имя объекта> : <имя класса> =
(<имя поля 1> : <значение поля 1>;
<имя поля 2> : <значение поля 2>;
. . .
<имя поля K>:<значение поля K>);

Способ инициализации полей объектов выбирают исходя из особенностей решаемых задач.

Пример 2.1. Описание класса (класс Окно). Пусть требуется описать класс, представляющий окно в текстовом режиме. Основным методом такого класса может быть процедура изображения окна, которая вызывает стандартные процедуры модуля Crt: Window, Textbackground и Clrscr. Целесообразно добавить два дополнительных метода, позволяющих определять размеры окна. В результате получим следующую программу:

```
Program WinOb;
Uses Crt;
Type
   {Описание структуры класса}
  Win = Object  {класс Окно}
        {Поля класса}
    X1,Y1,  {координаты верхнего левого угла окна}
    X2,Y2,  {координаты нижнего правого угла окна}
    Cf: byte;    {цвет фона}
        {Методы класса}
   Procedure Init(Xn1,Yn1,Xn2,Yn2,Cfn:byte); {инициализация полей}
   Procedure MakeWin;          {создание окна}
   Function GetSizeX : byte;  {определение размера окна по оси X}
   Function GetSizeY : byte;  {определение размера окна по оси Y}
   End;
  {Описание методов класса}
Procedure Win.Init;
  Begin X1:=Xn1; Y1:=Yn1; X2:=Xn2; Y2:=Yn2; Cf:=Cfn; End;
Procedure Win.MakeWin;
```

```
  Begin  Window(X1,Y1,X2,Y2); Textbackground(Cf); Clrscr End;
Function Win.GetSizeX;
   Begin   GetSizeX:=X2-X1+1 End;
Function Win.GetSizeY;
   Begin  GetSizeY:=Y2-Y1+1 End;
Var   Winvar : Win; {объявление объекта}
Begin
   with Winvar do    {для объекта Winvar выполнить}
      begin
       Init(1,1,80,25,1); {инициализировать поля объекта}
       MakeWin;          {вызвать метод изображения окна}
       Writeln(Cf,' - ,GetSizeX,' - ',GetSizeY); {вывести размеры окна}
      end;
Readkey;
End.
```

В результате работы программы на экране будет изображено синее окно, в которое будут выведены три числа: код цвета фона и размеры окна (1 - 80 - 25).

Доступ к полям и методам класса осуществляется с помощью оператора присоединения with. Можно было бы использовать с той же целью составные имена:

```
Winvar.Init(1,1,80,25,1);
Winvar.MakeWin;
Writeln(Winvar.Cf,' - ', Winvar.GetSizeX,' - ',Winvar.GetSizeY);
```

Параметр Self. Параметр Self является обобщенным именем экземпляра класса, которое неявно используется внутри методов для обращения к полям и методам. Компилятор автоматически обрабатывает данный параметр, поэтому явно его можно не указывать. Исключением является случай, когда какой-либо идентификатор текущего класса совпадает с внешним идентификатором, используемым в этом классе. Рассмотрим пример, в котором имеется необходимость в использовании параметра Self.

```
Type   Rec = Record             {тип записи}
         X,Y : word ;             {поля данных записи}
end;
Pos =Object          {объявление класса}
         X,Y : word;               {поля данных класса}
         Procedure Init(P : Rec);    {метод класса}
end;
Procedure Pos.Init;
  Begin
```

```
with P do    {присоединить имя записи P}
        begin
          {Передать поля данных записи полям данных класса}
          Self.X:= X;   Self.Y := Y ;
        end
  End;
Var  R:Rec;  P:Pos;
Begin  R.X:=1;  R.Y:=2;      {инициализировать поля записи}
        P.Init(R);          {инициализировать объект}
        Write( P.X, ' - ',P.Y); {вывести значения полей объекта}
End.
```

2.2. Наследование

В главе 1 сформулировано следующее определение: *Наследование* – это такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других классов (соответственно простое и множественное наследование).

В Borland Pascal 7.0 реализовано простое наследование, при котором у класса может быть только один родитель, но сколь угодно потомков.

Любой класс можно объявить потомком ранее описанного класса. Потомок наследует все данные и методы класса родителя и дополняет их своими данными и методами.

В результате использования механизма наследование отпадает необходимость заново описывать уже существующие в классе-родителе поля и методы. Требуется описать только те поля и методы, которых недостает в классе-родителе.

Доступ к полям, описанным в классе-родителе, осуществляется также как к собственным. Поиск метода в иерархии классов выполняется следующим образом.

1. В первую очередь компилятор устанавливает тип объекта.

2. Далее он ищет метод в классе объекта и если находит, то подключает его.

3. Если метод в классе объекта не найден, то идет поиск в классе-родителе. В случае успеха вызывается метод родителя.

4. Если метод в классе-родителе не найден, то поиск продолжается в классах-предках до тех пор, пока вызванный метод не будет найден.

Если компилятор не обнаруживает метод, то он фиксирует ошибку компиляции номер 44 (*Field identifier expected*– «ожидается имя поля или метода класса»).

Механизм наследования обеспечивает создание дерева родственных классов. Дерево может иметь несколько уровней и ветвей, где на каждом уровне добавляются необходимые поля и методы.

Структура описания класса-потомка:

Type <имя класса-потомка>=**Object(**<имя класса-родителя>**)**
<новые поля класса-потомка>
<новые методы класса-потомка>
End;

Примеры 2.2 и 2.3 иллюстрируют преимущества механизма наследования.

Пример 2.2. Разработка сложного класса без использования наследования (класс Символ). Пусть описан класс Символ (Symb), который позволяет создавать окно и выводить символ в заданную позицию.

```
Program SymbOb;
Uses Crt;
Type
  Symb = Object  {класс Символ}
          {Поля класса}
        X1,Y1, {координаты верхнего левого угла окна}
        X2,Y2, {координаты нижнего правого угла окна}
        Cf,Cs, {цвет фона и символов}
        Col,Line:byte; {позиция символа}
        Sym:char;      {символ}
          {Методы класса}
        Procedure Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln:byte;Sm:Char);
        Procedure MakeWin; {создание окна}
        Procedure Print;    {вывод символа}
     End;
Procedure Symb.Init;
Begin
  X1:=Xn1; Y1:=Yn1; X2:=Xn2; Y2:=Yn2; Cf:=Cfn;
  Cs:=Csn; Col:=Cl; Line:=Ln; Sym:=Sm;
End;
Procedure Symb.MakeWin;
Begin
  Window(X1,Y1,X2,Y2); Textbackground(Cf); Clrscr
End;
Procedure Symb.Print;
Begin
  TextColor(Cs); Gotoxy(Col,Line); Write(Sym)
End;
Var   Symbvar : Symb; {объявление экземпляра класса}
Begin
  Symbvar.Init(1,1,80,25,7,1,40,12,'A'); {инициализировать объект}
```

```
  Symbvar.MakeWin; {нарисовать окно}
  Symbvar.Print;   {вывести символ}
Readkey;
End.
```

Если сравнить примеры 2.1 и 2.2, то можно заметить, что поля X1, Y1, X2, Y2, Cf и метод MakeWin совпадают. Следовательно, класс Symb можно наследовать от класса Win.

Графически наследование классов можно изобразить в виде дерева иерархии родственных классов. При этом, в качестве комментария, могут быть указаны поля и методы классов.

Пример 2.3. Использование наследования (классы Окно и Символ). На рис. 2.1 представлена иерархия классов для рассматриваемого примера, в которой показано, что класс Символ наследуется от класса Окно. Справа для классов указаны используемые поля и методы, причем для класса-потомка указаны только те поля и методы, которые отсутствуют в родительском классе. Иерархия классов в данном случае простая и имеет всего два уровня, где Win считается классом первого уровня, а Symb – классом второго уровня.

В рассматриваемом примере 2.3 класс Symb объявлен как потомок класса Win.

```
Program WinSymbOb;
Uses Crt;
{.........................................}
{ Описание класса Win и его методов из примера 2.1 }
{.........................................}
Type
 Symb = Object(Win) {класс-потомок «символ»}
        {Новые поля}
      Cs, {цвет символов}
      Col,Line:byte; {позиция символа}
      Sym:char;      {символ}
```

Класс	Поля и методы
Win	Поля: X1, Y1, X2, Y2 – координаты окна Cf – цвет фона Методы: Init, MakeWin, GetSizeX, GetSizeY
Symb	Новые поля: Cs – цвет символов Col, Line – позиция символа Sym – символ Новые методы: Init, Print

Рис. 2.1. Иерархия классов примера 2.3

```
      {Новые методы}
      Procedure    Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln:byte;Sm:Char);
                                            {инициализация полей}
      Procedure Print;  {вывод символа}
    End;
Procedure Symb.Init;
Begin
  Win.Init(Xn1,Yn1,Xn2,Yn2,Cfn); {используем родительский метод}
  Cs:=Csn; Col:=Cl; Line:=Ln; Sym:=Sm;
End;
Procedure Symb.Print;
  Begin TextColor(Cs); Gotoxy(Col,Line); Write(Sym) End;
Var  Symbvar : Symb; {объявление экземпляра класса}
Begin
  with Symbvar do begin
  Init(1,1,80,25,7,1,40,12,'A');  {инициализировать объект}
  MakeWin;                         {изобразить окно}
  Print                            {вывести символ}
  end;
Readkey;
End.
```

Результаты работы программы данного примера аналогичны результатам работы программы примера 2.2. Применение механизма наследования позволило повторно не описывать поля и методы ранее созданного класса, в результате чего заметно упростилась разработка класса Symb. Кроме этого, дополнительно появилась возможность использовать родительские методы GetSizeX и GetSizeY класса Win, также ранее определенные в классе Win.

Присваивание объектов. Для объектов возможно выполнение операции присваивания. Смысл данной операции заключается в том, что происходит поэлементное присваивание содержимого полей одного объекта полям другого. Данная операция применима как к объектам одного класса, так и к объектам классов одной иерархии. Причем разрешается только объектам родительских классов присваивать значения объектов производных классов. Присваивание в обратном порядке не допускается, так как, если в классе-потомке объявляются дополнительные поля, то при присваивании объекту класса-потомка значения объекта родительского класса эти дополнительные поля остаются неопределенными.

Формат записи операции присваивания выглядит следующим образом:

<объект класса-родителя> := <объект класса-потомка>

Так для классов примера 2.3 возможно только присваивание полям объекта класса Win значений полей объекта класса Symb, например:

```
{...............................................}
{... Описание классов Win и Symb из примера 2.3 .......}
{...............................................}
Var    Winvar : Win;              {объект-родитель}
       Symbvar : Symb; {объект-потомок}
Begin  Symbvar.Init(1,1,80,25,7,1,40,12,'A');.
       Winvar := Symbvar;      {передать поля данных}
       Winvar.MakeWin;         {изобразить окно}
Write(Winvar.Cf); {на сером фоне будет выведено синим цветом число 7}
End.
```

Операция присваивания применима и к динамическим объектам, память под которые запрашивается из динамической области, например

```
{...............................................}
{ ... Описание классов Win и Symb из примера 2.3 ....... }
{...............................................}
Var    Winvar : ^Win;              {указатель на объект-родитель}
       Symbvar : ^Symb;            {указатель на объект-потомок}
Begin  New(Symbvar);             {выделить память под объект}
       Symbvar^.Init(1,1,80,25,7,1,40,12,'A');. {инициализировать}
       Winvar := Symbvar;        {передать указатель}
       Winvar^ := Symbvar^;      {передать содержимое}
       Winvar^.MakeWin;          {изобразить окно}
       Write(Winvar^.Cf);        {вывести символ}
       Dispose(Symvar);          {освободить память}
End.
```

Более подробно применение динамических объектов и объектов с динамическими полями рассмотрено в разделе 2.4.

2.3. Полиморфизм

Borland Pascal реализует механизмы простого и сложного полиморфизма.

Простой полиморфизм. Механизм простого полиморфизма обеспечивает возможность задания различных реализаций некоторого единого по названию метода для классов различных уровней иерархии. Одноименные методы в этом случае называют статическими полиморфными.

В ООП считается, что все реализации одноименных методов иерархии образуют полиморфный метод семейства классов. Каждый вариант реализации при этом представляет собой отдельный аспект полиморфного метода семейства классов. Конкретный аспект статического полиморфного метода

определяется типом объекта на этапе компиляции программы (*раннее связывание*).

Рассмотрим случай применения простого полиморфизма в языке Borland Pascal 7.0.

Пример 2.4. Применение простого полиморфизма. Пусть необходимо разработать программу, которая выводит в окно не один символ, а последовательность одинаковых символов. Для построения нужного класса используем уже имеющиеся классы Win и Symb: наследуем класс Lot от класса Symb. При этом необходимо заменить метод Print, описанный в классе Symb (рис. 2.2). Для объектов класса Lot он будет выводить не один символ, а количество символов, заданное в поле N этого класса.

```
Program WinSymbLot;
Uses Crt;
{..................................................}
{Описание класса Win и его наследника Symb из примера 2.3 ..}
{..................................................}
Type
  Lot = Object(Symb)  {класс-потомок Lot}
      N : Word;  {новое поле - количество символов}
      Procedure Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln:Byte;Sm:Char; Nk:word);
                                              {инициализация полей}
      Procedure Print ; {переопределенный метод вывода символа}
   End;
Procedure Lot.Init;
  Begin     Symb.Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln,Sm);
  N := Nk;  {инициализация нового поля}
  End;
```

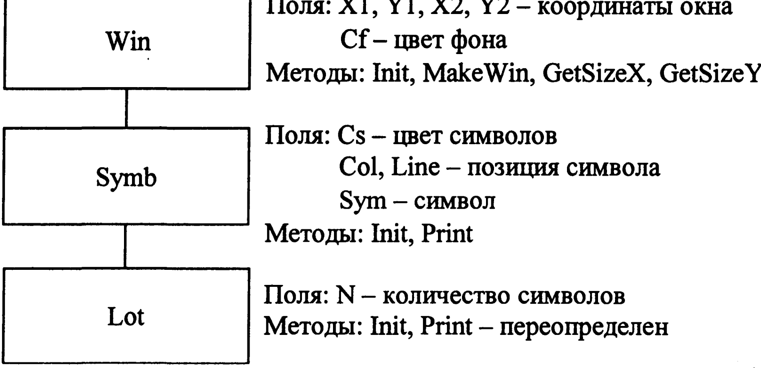


Рис. 2.2. Иерархия классов примера 2.4

```
Procedure Lot.Print;
Var  i:byte;
Begin
  for i:=1 to N do
     begin  Symb.Print; {вызвать метод родителя для вывода символа}
               inc(Col); inc(Line) {изменить позицию вывода}
     end;
End;
Var  Symbvar:Symb; {объявление объекта класса Symb }
       Lotvar : Lot;     {объявление объекта класса Lot }
Begin
  Symbvar.Init(1,1,80,25,7,1,40,12,'A');  {инициализировать поля}
  Symbvar.MakeWin;      {изобразить окно}
  Symbvar.Print;           {вывести символ – метод класса Symb}
  Readkey;
  Lotvar.Init(1,1,80,25,7,1,40,12,'B',10); {инициализировать поля}
  Lotvar.MakeWin;                          {изобразить окно}
  Lotvar.Print;                              {вывести символы – метод класса Lot}
  Readkey;
End.
```

По существу метод Print класса Symb и метод Print класса Lot – разные методы. Для объекта каждого класса вызывается метод, определенный в этом классе. По правилам Borland Pascal статические полиморфные методы могут иметь различные списки параметров в отличие от виртуальных полиморфных методов, рассматриваемых далее.

Сложный полиморфизм. При сложном полиморфизме конкретный аспект полиморфного метода также определяется типом объекта, для которого он вызван, но только не на этапе компиляции, а на этапе выполнения программы (*позднее связывание*).

Необходимость позднего связывания обусловлена в частности тем, что в Borland Pascal *разрешается указателю на объекты класса-родителя присваивать адрес объекта класса-потомка.* Поскольку при передаче объектов в качестве параметров используются указатели, формально описанному параметру типа некоторого класса может соответствовать фактический параметр не только того же типа, но и типа класса, *производного от указанного.*

В тех случаях, когда реальный тип объекта не определен, связывание объекта и метода на этапе компиляции программы приводит к тому, что при любых условиях (и для объектов родительских классов, и для объектов производных классов) вызывается родительский метод (раздел 1.6). Применение позднего связывания позволяет правильно определить требуемый аспект полиморфного метода.

Полиморфные методы, для которых применяется позднее связывание, называют *виртуальными.*

Для описания виртуальных методов используется служебное слово **virtual**:

Procedure <имя метода>(<список параметров>); **virtual**;
Function <имя метода> (<список параметров>):<тип функции>; **virtual**;

При использовании виртуальных методов должны выполняться следующие правила.

1. Если в некотором классе метод описан как виртуальный, то все производные классы, включающие метод с тем же именем, должны описать этот метод как виртуальный. Нельзя заменить виртуальный метод статическим.

2. Порядок расположения, количество и типы формальных параметров в одноименных виртуальных методах должны оставаться неизменными.

3. Класс, содержащий виртуальные методы, должен включать специальный статический метод – *конструктор.* Для этого метода слово *Procedure* в объявлении и реализации должно быть заменено словом *Constructor.* Конструктор неявно выполняет настройку механизма позднего связывания (обеспечивает связь объекта с ТВМ – раздел 1.6). *Этот метод должен быть вызван до первого обращения к виртуальному методу,* иначе происходит «зависание» компьютера. Как правило, конструктор используют для инициализации полей объекта и, соответственно, называют Init.

Вызов виртуальных методов осуществляется через ТВМ, что требует дополнительного времени. В результате программа будет работать медленнее, чем при использовании статических методов. Однако использование виртуальных методов позволяет существенно увеличить гибкость описания классов.

Фактически использование виртуальных методов обязательно в трех случаях:

1) полиморфный метод вызывается из статического метода родительского класса;

2) в программе используется процедура или функция с параметрами типа класса, которой в качестве фактического параметра может быть передан объект производного класса (*процедура с полиморфным объектом*), и эта функция вызывает полиморфный метод для объекта-параметра;

3) в программе используется указатель на базовый класс, которому может быть присвоен адрес объекта производного класса, и осуществляется вызов полиморфного метода для объекта по данному указателю.

Примечание. При разработке библиотек универсальных классов полиморфные методы часто объявляют виртуальными, исходя из возможных вариантов дальнейшего использования.

Продемонстрируем использование виртуальных методов, несколько изменив иерархию классов примера 2.4.

Пример 2.5. Вызов виртуальных методов из методов базового класса. Если при разработке иерархии классов Win – Symb – Lot учесть, что окно всегда рисуется для вывода символов, то в описание класса Win стоит включить метод Run, который изображает окно и выводит в него символы. Этот метод должен вызывать методы изображения окна MakeWin и вывода символов в окно Print. Метод MakeWin в классах-потомках не переопределяется (статический метод). Метод Print в классах Symb и Lot переопределяется (полиморфный метод). В классе Win метод Print не предусмотрен, поэтому добавим в этот класс абстрактный («пустой», не выполняющий никаких действий) метод Print.

Метод Run будет наследоваться классами Symb и Lot. Конкретный тип объекта, для которого вызывается метод Run, будет определен только на этапе выполнения, значит и нужный аспект полиморфного метода Print можно будет определить только на этапе выполнения программы, и, следовательно, метод Print должен быть объявлен виртуальным (случай 1 из рассмотренных выше).

Наличие в классах хотя бы одного виртуального метода требует объявления конструкторов классов. В качестве конструктора в данном примере будем использовать метод инициализации полей Init.

```
      Program MetVirt;
Uses Crt;
Type
 Win = Object   {родительский класс Окно}
      X1,Y1,X2,Y2,Cf: byte;
      Constructor Init(Xn1,Yn1,Xn2,Yn2,Cfn:byte);
      Procedure MakeWin;        {изображение окна}
      Function GetSizeX : byte;  {определение размера окна по X}
      Function GetSizeY : byte;  {определение размера окна по Y}
      Procedure Run;             {изображение окна с текстом}
      Procedure Print; virtual;  {абстрактный виртуальный метод}
    End;
Constructor Win.Init;
  Begin X1:=Xn1; Y1:=Yn1; X2:=Xn2; Y2:=Yn2; Cf:=Cfn; End;
Procedure Win.MakeWin;
  Begin Window(X1,Y1,X2,Y2); Textbackground(Cf); Clrscr End;
Function Win.GetSizeX;  Begin GetSizeX:=X2-X1+1 End;
Function Win.GetSizeY;  Begin GetSizeY:=Y2-Y1+1 End;
Procedure Win.Run;
  Begin MakeWin;  {вызов статического метода изображения окна}
        Print     {вызов аспекта виртуального полиморфного метода}
  End;
```

```
Procedure Win.Print; Begin   End; {резервный – «пустой» метод}
Type
  Symb = Object(Win)  {класс-потомок «символ»}
        Cs,Col,Line:byte;  Sym:char;
        Constructor Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln:byte;Sm:Char);
        Procedure Print; Virtual;   {вывод символа}
     End;
Constructor Symb.Init;
Begin  Win.Init(Xn1,Yn1,Xn2,Yn2,Cfn);
        Cs:=Csn; Col:=Cl; Line:=Ln; Sym:=Sm;
End;
Procedure Symb.Print;
Begin TextColor(Cs); Gotoxy(Col,Line); Write(Sym) End;
Type
  Lot = Object(Symb)  {класс-потомок Lot}
          N : Word;
          Constructor Init
                (Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln:Byte;Sm:Char; Nk:word);
          Procedure Print;  virtual;
     End;
Constructor Lot.Init;
  Begin  Symb.Init(Xn1,Yn1,Xn2,Yn2,Cfn,Csn,Cl,Ln,Sm);  N := Nk; End;
Procedure Lot.Print;
Var   i:byte;
  Begin for i:=1 to N do  begin  Symb.Print;  inc(Col); inc(Line) end;  End;
Var    V1:Win; V2:Symb;  V3:Lot;
Begin   V1.Init(20,5,40,15,12);                {конструировать объект V1}
        V2.Init(10,3,60,20,3,1,30,10,'A');     {конструировать объект V2}
        V3.Init(1,1,80,25,5,1,40,12,'B',10);   {конструировать объект V3}
        V1.Run; {вызвать «пустой» аспект метода Print}   Readkey;
        V2.Run; {вызвать метод Print класса Symb}         Readkey;
        V3.Run; {вызвать метод Print класса Lot}          Readkey;
End.
```

Использование процедур с полиморфным объектом предполагает передачу переменных типа класса в процедуру или функцию в качестве параметров. При этом описывается параметр родительского типа, а реально может быть передан объект не только родительского класса, но и любого из классов-потомков.

Пример 2.6. Использование процедуры с полиморфным объектом. Вместо метода Run можно описать процедуру с полиморфным объектом PolOb, в которой параметром является переменная типа Win. Такой подход также дает

возможность использовать процедуру PolOb, а в ней конкретно методы MakeWin и Print, для класса-родителя Win и всех его потомков. При этом нужный аспект полиморфного метода Print будет определен по типу переданного в процедуру параметра (V1,V2 или V3).

Если убрать слово virtual в методе родителя Print, то при обращении к любому потомку метод Print будет взят родительский (в данном случае «пустой»).

```
Program PrPolOb;
Uses crt;
Type
 Win = Object   {родительский класс Окно}
      X1,Y1,X2,Y2,Cf: byte;
      Constructor Init(Xn1,Yn1,Xn2,Yn2,Cfn:byte);
      Procedure MakeWin;          {изображение окна}
      Function GetSizeX : byte;  {определение размера окна по X}
      Function GetSizeY : byte;  {определение размера окна по Y}
      Procedure Print; virtual;  {абстрактный виртуальный метод}
    End;
Constructor Win.Init;
  Begin X1:=Xn1; Y1:=Yn1; X2:=Xn2; Y2:=Yn2; Cf:=Cfn; End;
Procedure Win.MakeWin;
  Begin Window(X1,Y1,X2,Y2); Textbackground(Cf); Clrscr End;
Function Win.GetSizeX;   Begin GetSizeX:=X2-X1+1 End;
Function Win.GetSizeY;   Begin GetSizeY:=Y2-Y1+1 End;
Procedure Win.Print;   Begin  End;
{..............................................}
{...... Описание классов Symb и Lot из примера 2.5.....}
{..............................................}
Var  V1:Win; V2:Symb; V3:Lot;  {экземпляры родственных классов}
    {Процедура с полиморфным объектом Obj}
Procedure PolOb(Var Obj: Win);
Begin
      Obj.MakeWin;  {вызвать обычный статический метод}
      Obj.Print     {вызвать виртуальный полиморфный метод}
End;
Begin
    V1.init(20,5,40,15,12);                {конструировать объект V1}
    V2.init(10,3,60,20,3,1,30,10,'A');    {конструировать объект V2}
    V3.init(1,1,80,25,5,1,40,12,'B',10); {конструировать объект V3}
    PolOb(V1); {выполнить с объектом V1}      Readkey;
    PolOb(V2); {выполнить с объектом V2}      Readkey;
    PolOb(V3); {выполнить с объектом V3}      Readkey;
End.
```

Механизм сложного полиморфизма необходимо использовать и при работе с объектами классов Win – Symb – Lot через указатели (случай 3 из рассмотренных выше). В этом варианте указателю на объект класса-родителя присваивается адрес объекта класса-потомка, а затем для него вызывается полиморфный метод или процедура с полиморфным объектом:

```
{.......................................................}
{....... Описание класса Win и его наследников............}
{.............. Symb и Lot из примера 2.5..................}
Type W=^Win; {указатель на родительский класс}
Var    V:W;  V2:Symb;  V3:Lot;
Procedure PolOb(Var Obj: W);  Begin  Obj^.MakeWin; Obj^.Print End;
Begin
        New(V); V1^.Init(20,5,40,15,5);
        V2.Init(10,3,60,20,3,1,30,10,'A');
        V3.Init(1,1,80,25,5,3,40,12,'B',10);
        PolOb(V1); Readkey;
        V1:=@V2;  {передать адрес 1-го потомка}
        PolOb(V1); {выводит: A} Readkey;
        V1:=@V3;  {передать адрес 2-го потомка}
        PolOb(V1); {выводит: BBBBBBBBBB} Readkey;
    End.
```

Тип полиморфного объекта в программе можно определить с помощью функции **TypeOf**:

TypeOf (<имя объекта>) : Pointer ;

или

TypeOf (<имя класса>) : Pointer ;

Данная функция возвращает указатель на ТВМ для конкретного объекта или класса. Она применима только к классам, включающим виртуальные методы.

Эта функция обычно используется для проверки фактического типа экземпляра, например:

if TypeOf(Self)=TypeOf(<ИмяЭкземпляра>*) then ... ;*

Чтобы определить фактический размер объекта, можно применить функцию **SizeOf**:

SizeOF (<имя объекта>): integer;

или

SizeOF (<имя класса>): integer;

Примечание. Для проверки корректности экземпляров программа должна компилироваться в режиме {*$R+*}. При этом генерируется вызов подпрограммы проверки правильности ТВМ перед каждым вызовом виртуального метода. Это замедляет работу программы, поэтому рекомендуется включать {*$R+*} только во время отладки.

По правилам Borland Pascal для инициализации полей статических объектов можно использовать типизированные константы (раздел 2.1). Следовательно, в программе примера 2.5 объекты можно было бы объявить следующим образом:

```
Const       V1:Win=(X1:20;Y1:5;X2:40;Y2:15;Cf:12);
                V2:Symb=(X1:10;Y1:3;X2:60;Y2:20;Cf:3;
                                  Cs:1;Col:30;Line:10;Sym:'A');
                V3:Lot=(X1:1;Y1:1;X2:80;Y2:25;Cf:5;
                                  Cs:1;Col:40;Line:12;Sym:'B';N:10);
```

Данный способ освобождает от необходимости вызова конструктора для инициализации объекта, содержащего виртуальные методы, так как в этом случае он активизируется автоматически.

2.4. Динамические объекты

Как уже указывалось ранее, описав класс, мы можем объявлять переменные этого класса, размещая их как в статической, так и динамической памяти. Для объявления динамического объекта используют указатели на объекты класса, описываемые следующим образом:

Var <имя указателя> : ^<имя класса>;

Для размещения объекта в динамической памяти используют процедуру или функцию New:

процедура
New (<имя указателя>);

функция
<имя указателя>:=**New**(<тип «указатель на класс»>) .

Если динамический объект использует виртуальные методы класса, то до обращения к этим методам он должен вызвать конструктор (раздел 2.3):

<имя указателя>^.<имя конструктора>(<список параметров>) .

В Borland Pascal 7.0 имеется возможность использования расширенного варианта процедуры и функции New, который позволяет в одной операции

выделить память под объект и вызвать конструктор:

процедура

New (<имя указателя>,
<имя конструктора>(<список параметров>));

функция

<имя указателя>: = **New**(<тип «указатель на класс»>,
<имя конструктора>(<список параметров>))

Такой вариант процедуры или функции New работает следующим образом. Сначала осуществляется попытка выделить память под объект, а затем вызывается конструктор.

В случае, если объект содержит динамически размещаемые поля, то выделение памяти под них обычно программируется в конструкторе. Для освобождения памяти, занимаемой динамическими объектами, используется стандартная процедура:

Dispose(<имя указателя>);

Если класс объекта включает виртуальные методы, то при уничтожении объекта обязательно должен вызываться специальный метод – *деструктор*. Так же, как для конструктора, для деструктора служебное слово Procedure должно заменяться служебным словом Destructor, что подразумевает выполнение некоторых специальных действий при уничтожении объекта: деструктор определяет реальный размер объекта и передает его процедуре Dispose. Если конструктор рекомендуется называть Init, то деструктор обычно называют Done. Деструкторы могут наследоваться, быть статическими или виртуальными. Обычно они используются для освобождения памяти, выделенной под динамические поля объекта.

Деструктор можно вызвать в расширенном формате процедуры Dispose:

Dispose (<имя указателя>,<имя деструктора>);

Для динамических объектов имеет смысл объявлять зарезервированный виртуальный деструктор, пусть даже и пустой:

Destructor <имя класса> . *Done;* ***virtual;***
Begin End;

Контроль распределения памяти. Если памяти для размещения динамического объекта или динамических полей окажется недостаточно, то вызов процедуры или функции New приведет к возникновению ошибки времени выполнения с номером 203 (Heap overflow error – «переполнение динамической памяти»).

Аварийного завершения программы в данном случае можно избежать, определив собственную функцию обработки данной ошибки и указав ее адрес среде программирования Borland Pascal:

```
Function HeapFunc(size:Word):integer; far;
        Begin HeapFunc:=1; end;
              . . .
HeapError:=@HeapFunc;
```

Тогда при обнаружении нехватки памяти процедура или функция New вернет указатель со значением nil, что можно проверить в программе.

Если такую ситуацию обнаруживает конструктор при распределении памяти для размещения динамических полей, то целесообразно, чтобы он отменил все уже выполненные распределения памяти. Для этого введена стандартная процедура Fail без параметров. Процедура Fail может быть вызвана только из конструктора. Она отменяет уже выполненные назначения памяти и возвращает значение nil в качестве адреса объекта.

В примере 2.7 показано, как можно организовать динамический объект с динамическими полями.

Пример 2.7. Динамический объект с динамическим полем и контролем выделения памяти. Разработаем класс WinD, который по-прежнему включает в себя поля и методы для создания окна с заданными цветами фона и символов, но для хранения параметров объекта использует массив (вектор атрибутов окна), память под который выделяется динамически. Данный массив состоит из 6 элементов, где элементы с индексами 1 – 4 – координаты окна, а элементы с индексами 5 и 6 – цвет фона и цвет символов окна.

```
Program DinObj;
Uses Crt;
Type
  APtr = ^Vaw;  {объявить указатель на массив}
  Vaw = array [1..6] of byte;
  WinPtr = ^WinD; {указатель на объект класса}
  WinD = Object
      A:APtr ;                      {указатель на вектор атрибутов окна}
      Constructor Init(An:Vaw);     {конструктор}
      Destructor Done; virtual;     {деструктор}
      Procedure MakeWin;            {создание окна}
      Procedure ColorWin;           {установка цвета фона и символов}
  End;
Constructor WinD.Init;
Begin
```

```
  New(A); {разместить поле A в динамической памяти}
  if A=nil then {при неудаче освободить память}
    begin WinD.Done; Fail end;
  A^:=An; {инициализация массива атрибутов}
End;
Destructor WinD.Done;
  Begin if A<>nil then Dispose (A); {освобождаем поле A} End;
Procedure WinD.MakeWin; {метод создания окна}
  Begin Window(A^[1],A^[2],A^[3],A^[4]) End;
Procedure WinD.ColorWin;
  Begin TextbackGround(A^[5]); TextColor(A^[6]); Clrscr End;
Function HeapFunc(size:Word):integer; far; Begin HeapFunc:=1; end;
Var   V :WinPtr ; {указатель на динамический объект}
Const
  Am:Vaw=(1,1,80,25,4,1); {координаты верхнего левого и нижнего
                          правого угла окна, цвет фона и символов}
Begin
   HeapError:=@HeapFunc; {подключить функцию обработки ошибки}
   Clrscr;
   New(V,Init(Am));        {разместить динамический объект}
   if V=nil then Halt(2); {если неудачно, то выйти}
   V^.MakeWin;      {вызвать методы для динамического объекта}
   V^.ColorWin;
   Dispose(V,Done); {уничтожить объект и освободить поля}
   Readkey;
End.
```

Для контроля размещения динамических полей *статического* объекта осуществляется проверка возвращаемого значения конструктора. Если это значение равно **false**, то в конструкторе была выполнена директива **fail**.

Пример 2.8. Статический объект с динамическим полем и контролем выделения памяти.

```
Program DinObj;
{...................................................}
{....... Описание класса WinD и процедуры обработки .....}
{.............. ошибок из примера 2.7......................}
Var   W :WinD ; {статический объект}
Const
  A:Vaw=(1,1,80,25,4,1); {координаты верхнего левого и нижнего
                          правого угла окна, цвет фона и символов}
```

```
Begin
   HeapError:=@HeapFunc;  {подключить функцию обработки ошибки}
   Clrscr;
   if not W.Init(A) then Halt(2);  {если неудачно, то выйти}
   W.MakeWin;    {вызвать методы для динамического объекта}
   W.ColorWin;
   W.Done; {освободить память, отведенную под динамическое поле}
   Readkey;
End.
```

Существуют задачи, для которых вопрос управления памятью является очень важным.

Пример 2.9. Использование динамических объектов (программа «Снежинки»). Допустим, требуется создать программу, в которой случайным образом генерируются объекты в виде разноцветных снежинок (такая программа может выполнять функцию заставки на экране компьютера). Необходимо предусмотреть возможность изменения состояния объектов, в частности: снежинки должны перемещаться по экрану, менять свои размеры и, достигнув определенного положения, исчезать (рис. 2.3).

При объектной декомпозиции мы получаем три типа объектов: псевдообъект – Основная программа и множество объектов – снежинок, которые делятся на Изменяющиеся снежинки и Падающие снежинки (рис. 2.4).

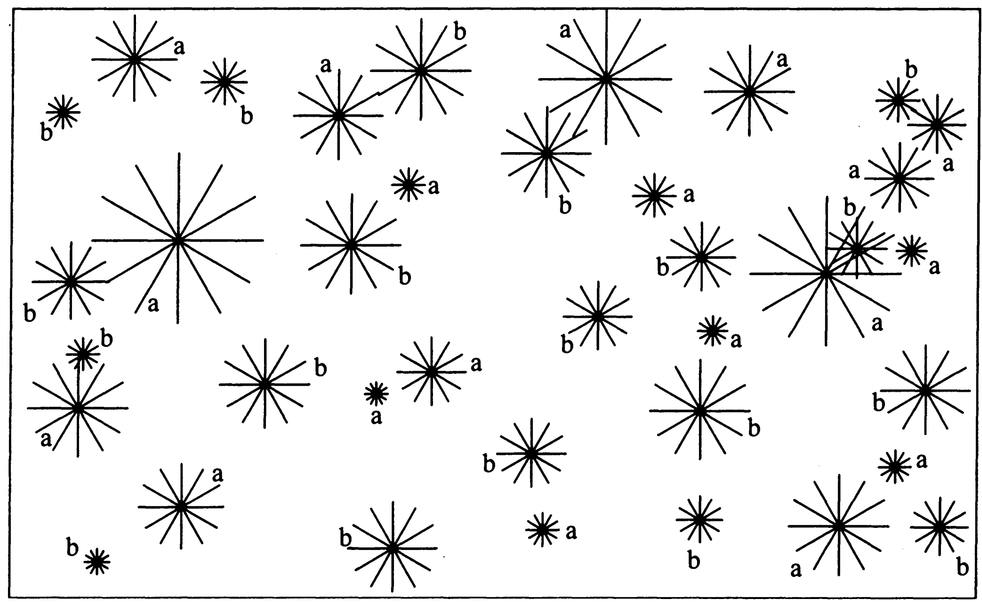


Рис. 2.3. Вид экрана при выполнении программы «Снежинки»:
а – снежинка меняет размер; b – снежинка перемещается вниз

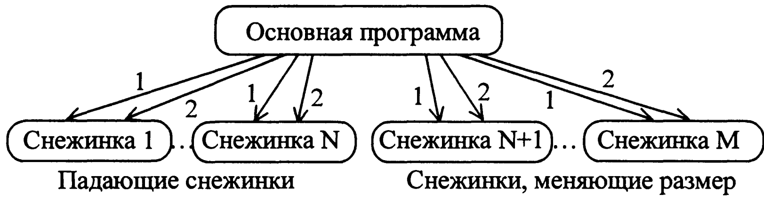


Рис. 2.4. Объектная декомпозиция для программы «Снежинки».

Примечание. Псевдообъект Основная программа имеет смысл вводить при выполнении объектной декомпозиции программных систем, которые будут реализовываться в среде MS DOS, для конкретизации источника сообщений, получаемых объектами предметной области задачи.

Для реализации двух типов объектов Снежинок построим иерархию классов (рис. 2.5). Класс Snow содержит основные поля и методы, используемые для изображения снежинок. Он является родительским классом, а классы Snow1 и Snow2 – его потомками.

Объекты – снежинки получают сообщения «Создать и инициализировать» и «Изменить внешний вид». Соответственно классы снежинок должны содержать методы обработки этих сообщений. Для инициализации полей реализован метод Init. Определения закона изменения внешнего вида снежинки осуществляется в специальном методе Drag. Закон изменения размера снежинки описан в методе Drag класса Snow1, а закон перемещения по экрану в одноименном методе класса Snow2.

В программе инициализируется массив указателей на динамические объекты – снежинки разных типов, т.е. каждый раз при вызове полиморфного метода Drag должен определяться требуемый аспект, соответствующий типу реально созданного объекта. Следовательно, метод Drag должен быть объявлен виртуальным (3-й случай – раздел 2.3). Этот метод также должен быть объявлен в базовом классе, так как иначе компилятор будет возражать против его вызова

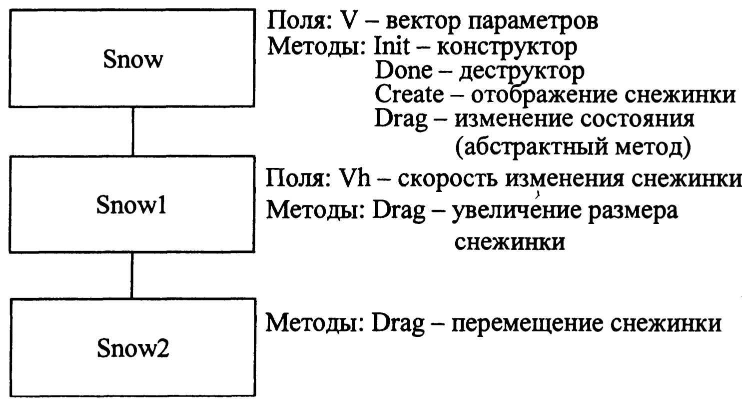


Рис. 2.5. Иерархия классов программы «Снежинки»

по указателю на объект базового класса. Поскольку реальное наполнение данного метода выполняется в других классах, в базовом класса метод Drag описывается абстрактным («пустым»).

```
Program SnowOb;
Uses graph,dos,crt;
Type
  Ptr = ^Vt; {указатель на массив}
  Vt = array [1..6] of integer; {вектор параметров снежинки}
  Snow = Object               {родительский класс Снежинка}
        V : Ptr;             {вектор параметров снежинки}
        Constructor Init;       {конструктор}
        Destructor  Done; virtual;     {деструктор}
        Procedure  Create;  {метод создания снежинки}
        Procedure  Drag;virtual;   {абстрактный метод}
       End;
Constructor Snow.Init; Var c,i:word;
Begin
  New(V);               {выделение памяти для поля данных }
  V^[4]:=Random(14)+1;  {цвет снежинки}
  V^[5]:=Random(30)+20; {максимальный радиус (размер снежинки)}
  V^[6]:=1;             {флаг – «объект включен»}
  Gettime(i,i,i,c);        { использование сотых долей секунды}
  for i:=0 to c do
    begin     {определение координат центра и радиуса снежинки}
     V^[1]:=Random(GetmaxX);   V^[2]:=Random(GetmaxY);
     V^[3]:=Random(20)+5;
    end;
End;
Destructor Snow.Done; {освобождение поля V}
  Begin   if V<>nil then Dispose(V)   End;
Procedure Snow.Create;
Var u:real;
Begin
   u:=0; Setcolor(V^[4]);
   repeat  {нарисовать снежинку из 12 лучей}
         Line(V^[1],V^[2],Round(V^[1]+V^[3]*cos(u)),
                          Round(V^[2]+V^[3]*sin(u)));
     u:=u+pi/6;
   until u>2*pi;
End;
Procedure   Snow.Drag;
Begin End;
```

```
Type
   Snow1 = Object(Snow) {первый потомок}
       Vh:byte; { скорость изменения состояния снежинки}
       Constructor Init(Vhn:Byte);
       Procedure  Drag; virtual; {изменение состояния снежинки}
    End;
Constructor Snow1.Init;  Begin Snow.Init; Vh:=Vhn End;
Procedure  Snow1.Drag;
Var c:byte;
Begin
    c:= V^[4]; V^[4]:=0; {изменение цвета пера на цвет фона}
    Create ;         { изображение снежинки цветом фона – стирание}
    V^[4]:=c;        {восстановление цвета пера}
    if (V^[3] < V^[5]) and (V^[2]+V^[5] < GetmaxY) then {если снежинка не
                                               превысила максимального размера}
      begin inc(V^[3],Vh);  Create { то увеличение снежинки}end
    else V^[6]:=0;     {иначе установка флага «объект отключен»}
End;
Type
  Snow2 = Object(Snow1)                {второй потомок}
       Procedure  Drag; virtual;  {переопределенный метод}
     End;
Procedure  Snow2.Drag;
Var c:byte;
Begin
  c:= V^[4]; V^[4]:=0; {изменение цвета пера}
  Create ; { стирание снежинки}
  V^[4]:=c; {восстановление цвета пера}
  if  V^[2]+V^[5] < GetmaxY then   {если снежинка не достигла пределов
                                                            экрана}
    begin inc(V^[2],Vh); { то смещение снежинки} Create   end
  else V^[6]:=0;   {иначе установка флага «объект отключен»}
End;
Type    Sd1 =^Snow1;     {указатель на первый потомок}
        Sd2 =^Snow2;     {указатель на второй потомок}
Const N=100;  {максимальное количество снежинок}
Var  S : array[0..N] of ^Snow; {описание массива указателей}
       a,r,i,k,m:integer;  {вспомогательные переменные}
Begin
    a:=Detect; Initgraph(a,r,'D:\BP\BGI');randomize;
    Setbkcolor(Black); {черный фон экрана}
    k:=40; {количество снежинок на экране}
    for i:=0 to k do {инициализировать и вывести объекты}
```

```
        begin m:=random(2);   {определить тип снежинки}
              if m=1 then  S[i]:=New(Sd1,Init(1)) else S[i]:=New(Sd2,Init(1));
              S[i]^.Create;
        end;
    repeat  { цикл изменения объектов}
       for i:=0 to k do
         begin S[i]^.Drag;  {изменить снежинку}
              if S[i]^.V^[6]=0 then  {если флаг «объект отключен»}
                begin
                  Dispose(S[i],Done); {уничтожить объект}
                  if random(2)=0 then  S[i]:=New(Sd1,Init(1))
                  else S[i]:=New(Sd2,Init(1)); {создать новый}
                  S[i]^.Create;          {нарисовать снежинку}
                end;
         end;
    until Keypressed;
    for i:=1 to k do  {уничтожить оставшиеся объекты}
    Dispose(S[i],Done);
End.
```

2.5. Создание библиотек классов

При создании библиотек классов целесообразно скрыть детали реализации классов. В этом случае описание классов можно выполнить в интерфейсном разделе модуля, а тела методов определить в разделе реализации. Также можно описывать внутренние классы, которые полностью определяются в разделе реализации. В свою очередь класс, определенный в интерфейсном разделе модуля, может иметь потомков в разделе реализации модуля.

В случае, когда модуль В использует модуль А, модуль В может определять производные классы от любого класса, экспортируемого модулем А.

Созданные модули могут поставляться в виде подключаемых (.tpu) файлов с распечаткой классов, их полей и методов, определенных в интерфейсном разделе модуля. Пользователи такого модуля могут, используя механизмы наследования и полиморфизма, создавать на его основе новые классы.

Библиотека может экспортировать не только классы, но и объекты этих классов. Если необходимо экспортировать объекты, содержащие виртуальные методы, то для таких переменных в разделе инициализации можно разместить вызовы конструкторов.

С помощью стандартной директивы **private** можно объявить часть полей и методов скрытыми от пользователей модуля. *Они будут доступны только внутри модуля, где объявлен класс.*

Все, что объявлено после директивы private, становится недоступным извне. Использование только данной директивы накладывает ограничения на

последовательность описания классов, где используются как общедоступные, так и скрытые поля и методы.

Чтобы описание класса сделать более гибким, используют директиву **public**. Данная директива делает поля и методы класса общедоступными вне модуля Unit.

Структура модуля с описанием класса и экспортируемых объектов выглядит следующим образом:

```
    Unit <имя модуля>;
    Interface  {интерфейсный раздел}
    Type
            <имя класса> = object
               private  <скрытые поля>;
               public    <доступные поля>;
               private  <скрытые методы>;
               public    <доступные методы>;
            End;
         Var <объявление экспортируемых объектов класса>
  Implementation  {раздел реализации}
   {реализация методов}
  Begin  {раздел инициализации}
   {вызовы конструкторов экспортируемых объектов}
  End.
```

Рассмотрим пример, в котором описание класса размещается в модуле.

Пример 2.10. Размещение описания класса в модуле. Для демонстрации используем пример 2.7, в котором:

1) описание класса вынесем за пределы основной программы в отдельный модуль *Unit*;

2) с помощью директивы private скроем динамическое поле А (вектор атрибутов окна) метод ColorWin класса WinD.

Следует иметь в виду, что только после создания модуля Obj_unit.tpu скрытые поля и методы с помощью директивы private будут недоступны пользователям.

```
Unit Obj_Unit;
Interface {интерфейсный раздел}
Uses Crt;
Type
  Vaw = array [1..6] of byte;
  APtr = ^Vaw;  {указатель на массив}
  WinPtr = ^WinD; {указатель на объект}
```

```
   WinD = Object
      private {скрытое поле} A:APtr; {вектор атрибутов окна}
      public {общедоступные методы}
         Constructor Init(An:Vaw);  {конструктор}
         Destructor Done; Virtual;  {деструктор}
         Procedure MakeWin;         {изображение окна}
      private {скрытый метод}
         Procedure ColorWin; {установка цвета фона и символов}
   End;
Implementation  {раздел реализации методов}
Constructor WinD.Init;
  Begin  New(A); if A=nil then begin  WinD.Done; Fail end;
         A^:=An;
  End;
Destructor WinD.Done;
  Begin  if A<>nil then Dispose (A);  End;
Procedure WinD.MakeWin;
  Begin Window(A^[1],A^[2],A^[3],A^[4]); Self.ColorWin End;
Procedure WinD.ColorWin;
  Begin TextbackGround(A^[5]); TextColor(A^[6]); Clrscr  End;
End.
```

Текст основной программы, использующий модуль ObjUnit:

```
Program ModOb;
Uses ObjUnit;               {подключение созданного модуля}
Var
      V : WinPtr ;  {указатель на динамический объект}
      A : Vaw;      {вектор атрибутов окна}
Begin
      A[1]:=1; A[2]:=1; A[3]:=80; A[4]:=25; {координаты окна}
      A[5]:=7; A[6]:=1;                     {цвет фона и символов}
      New(V,Init(A));     {размещение динамического объекта}
      if V=nil then Halt(2);
      V^.MakeWin; {применение доступных методов}
      Dispose(V,Done)   {освобождение объекта и полей}
End.
```

Попытка обратиться непосредственно к полям класса с помощью составных имен, например для их инициализации:

```
V^.A^[1]:=1;    V^.A^[2]:=1;  V^.A^[3]:=60;
V^.A^[4]:=20;  V^.A^[5]:=3;   V^.A^[1]:=2;
```

приведет к ошибке 44 (*Field identifier expected* – «ожидается имя поля или метода класса»). Обусловлено это тем, что стандартная директива private делает недоступными поля класса WinD из основной программы примера 2.7.

Аналогичная ошибка обнаружится при прямом обращении:

V^.ColorWin;

Метод ColorWin будет также недоступен из основной программы, как и поля класса Win. С другой стороны, данный метод можно использовать внутри модуля, в частности в методе MakeWin.

2.6. Композиция и наполнение

Для реализации объектов, находящихся между собой в отношении включения (см. рис. 1.15), могут использоваться механизмы наследования или композиции.

Композицией называется такое отношение между классами, когда один является частью второго.

Композиция реализуется включением в класс объектных полей и применяется в тех случаях, когда наследование по каким-либо причинам невозможно или нецелесообразно.

Borland Pascal 7.0 предоставляет возможность использования в классах объектных полей. Рассмотрим это на примере.

Пример 2.11. Использование объектных полей. Рассмотрим программу, в которой класс Выводимая строка (OutS) объявлен как часть класса Окно (Win). Это реализовано включением в структуру класса Win объектного поля Ob_Field.

```
Program Compos;
Uses Crt;
Type
 OutS = Object {класс Выводимая строка}
      S:string; {строка вывода}
      Cs:byte; {цвет символов строки}
      Procedure Init(Sn:string;Csn:byte); {инициализация полей}
      Procedure Print; {вывод строки}
  End;
Procedure OutS.Init; Begin S:=Sn; Cs:=Csn End;
Procedure OutS.Print; Begin Textcolor(Cs); Write(S) End;
Type
 Win = Object {класс Окно}
      X1,Y1,X2,Y2, {координаты окна}
      Cf:byte; {цвет фона}
```

```
        Ob_Field : OutS; {статическое объектное поле}
        Procedure Init(Xn1,Yn1,Xn2,Yn2,Cfn:byte;Obn:OutS);
        Procedure MakeWin;   {изображение окна}
    End;
Procedure Win.Init;
Begin  X1:=Xn1; Y1:=Yn1; X2:=Xn2; Y2:=Yn2; Cf:=Cfn;
        Ob_Field:=Obn;  {инициализация объектного поля}
End;
Procedure Win.MakeWin;
Begin
   Window(X1,Y1,X2,Y2); Textbackground(Cf); Clrscr;
   Ob_Field.Print; {обращение к методу Print класса OutS как к методу поля
                                          класса Win}
End;
Var    V:Win; F:OutS; {объявление экземпляров классов}
Begin
        F.Init('Объектное поле',1); {инициализировать объекты}
        V.Init(20,10,35,10,0,F);    {F – параметр-объект}
        V.MakeWin; {изобразить окна и вывести строку}
       Readkey;
End.
```

Наполнение отличается от композиции тем, что вместо объектных полей используются поля, содержащие указатели на объекты.

Пример 2.12. Использование полей – указателей на объекты. Рассмотрим программу, в которой автоматически генерируется заданное количество окружностей (объектов), перемещающихся по линейному закону в пределах определенной зоны экрана (рис. 2.6).

В программе осуществляется анализ на столкновение всех окружностей друг с другом и с зоной. При столкновении окружности меняют направление движения, «оттолкнувшись» друг от друга.

Диаграмма объектов для данного примера представлена на рис.2.7. Для реализации объектов программы потребуются классы: Zona (зона) и Cir (окружность). Класс Zona будет включать поля, определяющие размер зоны на экране, поле N – для хранения количества окружностей и поле P – массив указателей на объекты класса Cir. Он также будет включать методы инициализации Init, изображения зоны Out и перемещения окружностей Run. Удобно определить внутренний метод Control, который будет определять наличие столкновений всех созданных объектов-окружностей друг с другом и зоной (рис.2.8).

Класс Cir существенно проще. Он должен содержать поля, определяющие его местонахождение, радиус и цвет, а также метод перемещения и внутренний

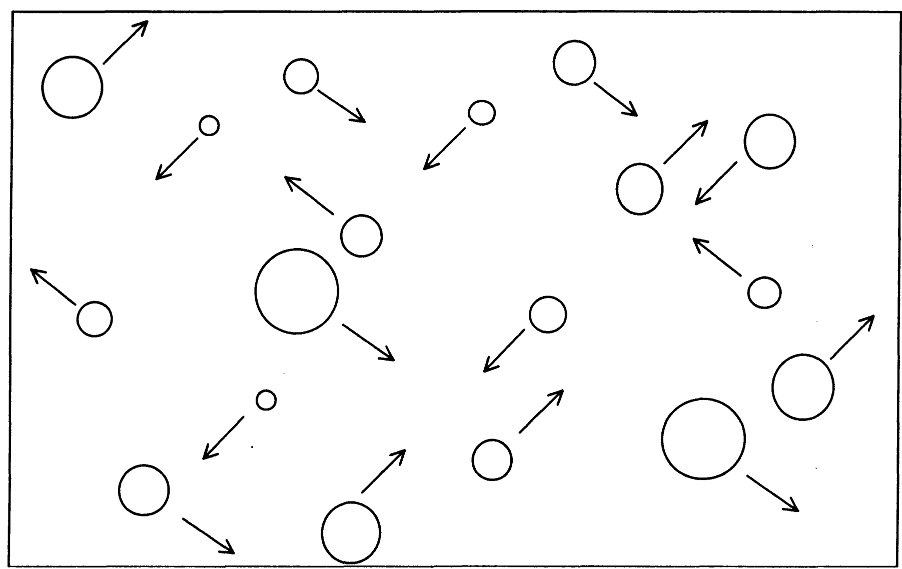


Рис. 2.6. Вид экрана при выполнении программы «Окружности»

метод изображения. Запрос зоны о местоположении окружности можно реализовать простым обращением к соответствующим полям объекта.

Рассмотрим работу программы, акцентируя внимание на использовании полей указателей на объекты. Вначале задаются параметры зоны и определяется количество размещаемых окружностей. Далее инициализируются объекты типа Cir и объект типа Zona. В результате полю Р объекта Z будет передан массив указателей на объекты типа Cir.

В методе Control объекта Z отслеживается ситуация возможного столкновения друг с другом всех созданных экземпляров классов. Если прогноз на столкновение каких-либо объектов для следующего шага положительный, то метод Control изменяет направление движения окружностей.

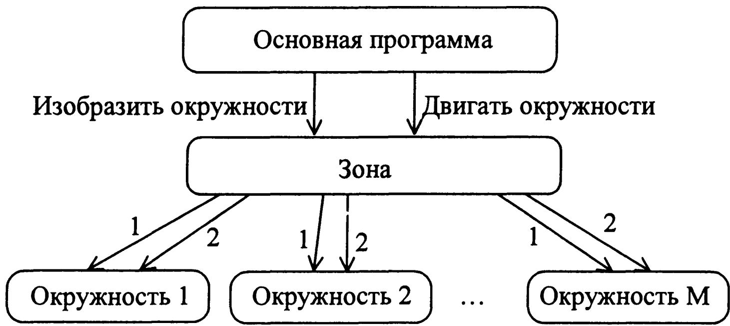


Рис. 2.7. Диаграмма объектов программы «Окружности»
1 – местоположение и размер окружности; 2 – переместить окружность

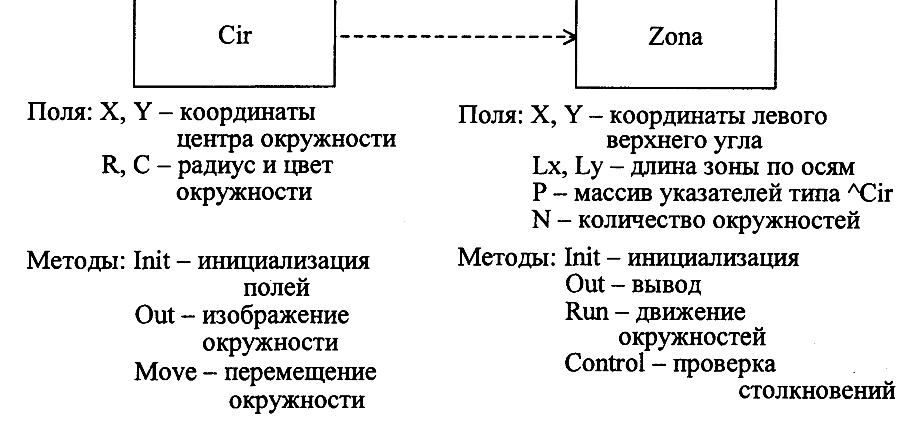


Рис. 2.8. Диаграмма классов программы «Окружности»

Такое управление становится возможным благодаря использованию указателей, переданных объекту типа Zona.

```
Program MoveCir;
Uses graph,crt;
Type
   Cp=^Cir;            {указатель на класс}
  Cir=Object           {класс Окружность}
     X,Y,R,C,          {координаты центра, радиус и цвет окружности}
     Dx,Dy:integer;    {шаг перемещения}
     Constructor Init(Xn,Yn,Rn,Cn,Dxn,Dyn:integer); {конструктор}
     Procedure Out;  {изображение окружности}
     Procedure Move; {перемещение окружности}
   End;
Constructor Cir.Init;
  Begin R:=Rn; C:=Cn; Dx:=Dxn; Dy:=Dyn; X:=Xn; Y:=Yn End;
Procedure Cir.Out;
  Begin Setcolor(C); Circle(X,Y,R) End;
Procedure Cir.Move;
 Var cc:integer;
 Begin      cc:=C; C:=Getbkcolor; Out; C:=cc; X:=X+Dx; Y:=Y+Dy; Out;
 End;
Type
  mascp=array [1..100] of Cp; {массив указателей}
  Zona=Object     {класс Зона}
     X,Y,Lx,Ly,     {координаты верхнего левого угла и размеры зоны}
     C:word;            {цвет зоны}
     P:mascp;           {поле – массив указателей}
     N:byte;             {количество окружностей}
```

```
      Constructor Init(Xn,Yn,Lxn,Lyn,Cn:word;Pn:mascp;Nn:byte);
      Procedure Out;      {изображение зоны}
      Procedure Control; {проверка объектов на столкновение}
      Procedure Run;      {перемещение окружностей}
   End;
 Constructor Zona.Init;
   Begin X:=Xn; Y:=Yn; Lx:=Lxn; Ly:=Lyn; C:=Cn; P:=Pn; N:=Nn End;
 Procedure Zona.Out;
   Begin  Setcolor(C); Rectangle(X,Y,X+Lx,Y+Ly)  End;
 Procedure Zona.Control;
    {Прогноз на столкновение двух окружностей}
   Procedure CrCr(P1,P2:Cp);
     Var wx,wy:real;
      Begin    wx:=P1^.X+P1^.Dx-P2^.X-P2^.Dx;
               wy:=P1^.Y+P1^.Dy-P2^.Y-P2^.Dy;
               if (sqrt(sqr(wx)+ sqr(wy)) <= P2^.R+P1^.R) then
          begin {смена направления перемещения}
                       P1^.Dx:=-P1^.Dx; P2^.Dx:=-P2^.Dx;
                       P1^.Dy:=-P1^.Dy; P2^.Dy:=-P2^.Dy;
         end;
      End;
     {Прогноз на столкновение окружности с зоной}
   Procedure CrZn(P:Cp);
     Begin
        if (P^.X+P^.Dx-P^.R <= X) or (P^.X+P^.Dx+P^.R >= X+Lx)
          then    P^.Dx:=-P^.Dx; {смена направления движения по X}
        if (P^.Y+P^.Dy-P^.R <= Y) or (P^.Y+P^.Dy+P^.R >= Y+Ly)
          then   P^.Dy:=-P^.Dy; {смена направления движения по Y}
     End;
 Var i,j:integer;
 Begin
    {Проверка на столкновение всех окружностей друг с другом}
   for i:=1 to N-1 do  for j:=i+1 to N do  CrCr(P[i],P[j]);
    {Проверка на столкновение с зоной всех окружностей}
   for i:=1 to N do CrZn(P[i]);
 End;
 Procedure Zona.Run;
  Var k:integer;
  Begin Control;  {вызвать метод проверки на столкновение}
          for k:=1 to n do  P[k]^.Move;
  End;
                { основная программа }
 Var    Z:Zona;   {объект класса Zona}
```

```
      V:mascp; {массив указателей на класс Окружность}
      a,rg,k,d,i, {вспомогательные переменные}
      x,y,lx,ly,lmin,cz, {параметры зоны}
      n,nmax,   {max и текущее количество окружностей}
      xc,yc,cc, {координаты центра и цвет окружности}
      r,rm,     {текущий и максимальный радиус}
     dx,dy:integer; {шаг перемещения окружности по осям}
Begin
  Clrscr; Writeln('Введите: ':20);
  Write('координату верхнего левого угла зоны по X - ');  readln(x);
  Write('координату верхнего левого угла зоны по Y - ');  readln(y);
  Write('размер зоны по оси X - ');  readln(lx);
  Write('размер зоны по оси Y - ');  readln(ly);
  cz:=random(14)+1; {цвет зоны}
  if lx<=ly then lmin:=lx else lmin:=ly;
  rm:=random(round(lmin/2))+3; {max радиус}
  nmax:=sqr(round(lmin/(2*rm+1))); if nmax>100 then nmax:=100;
  Write('количество окружностей (не более ',nmax,') - '); readln(n);
  Write('задержку - '); readln(d);
   xc:=x+rm+1; yc:=y+rm+1; k:=n;
  while k>0 do
    begin {инициализация окружностей}
      r:=random(rm)+1; cc:=random(15)+1;
      repeat dx:=random(5)-3 until dx<>0;
      repeat dy:=random(5)-3 until dy<>0;
      if (xc+rm+1) < (x+lx) then
         begin New(V[k]); V[k]^.Init(xc,yc,r,cc,dx,dy); {окружности}
                dec(k); xc:=xc+2*rm+1
         end
      else  if (yc+rm+1)< (y+ly) then
           begin yc:=yc+2*rm+1; xc:=x+rm+1 end;
  end;
  a:=Detect; Initgraph(a,rg,'C:\TP\BGI');
 Z.Init(x,y,lx,ly,cz,V,n);        {инициализировать зону}
 Z.Out;                           {отобразить зону}
 repeat
    Z.Run;        {вызвать метод движения окружностей}
    for i:=1 to 10  do delay(d);
 until Keypressed;
 Closegraph;
End.
```

2.7. Разработка программ с использованием объектно-ориентированного программирования

Поэтапно рассмотрим процесс разработки программы с использованием технологии ООП.

Пример 2.13. Программа «Текстовые эффекты». Условие задачи. Разработать программу, которая создает на экране компьютера «бегущие» строки с разными траекториями движения (рис. 2.9). Предусмотреть возможность добавления новых функций управления строками без изменения ранее разработанного текста программы.

На первом этапе необходимо выполнить анализ и объектную декомпозицию предметной области задачи.

В языках программирования обычно под строкой понимают последовательность символов. Для отображения на экране «бегущих» строк со сложными траекториями движения необходимо иметь возможность управления каждым символом строки (объект Символ). Такое управление должно включать возможность создания Символов и отображения их при перемещении в соответствии с заданным законом. Для управления символами

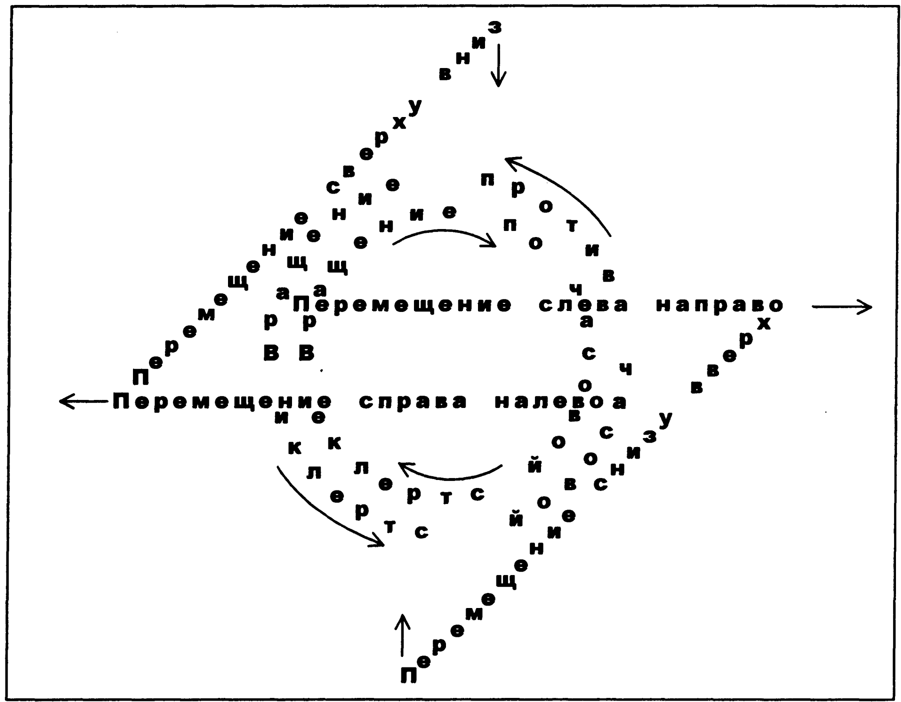


Рис. 2.9. Вид экрана при выполнении программы «Текстовые эффекты»

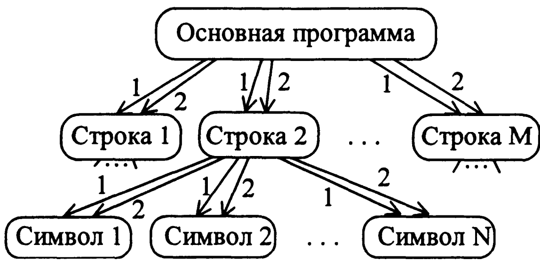


Рис. 2.10. Диаграмма объектов программы «Текстовые эффекты»:
1 – создать; 2 – вывести

строки необходим специальный объект, который, в соответствии с отображаемым предметом, назовем Строкой (рис. 2.10). Каждая строка будет получать сообщения «Создать» и «Вывести на экран». Конкретный закон перемещения символов строки будет определяться отдельно для каждой строки при разработке соответствующего класса.

Второй этап включает:

– логическое проектирование – разработку классов для реализации данной системы;

– физическое проектирование – разбиение программы на модули.

Разрабатывая классы для реализации объектов, полученных в результате объектной декомпозиции, необходимо ответить на следующие вопросы.

1. Сколько классов требуется для реализации всех объектов, полученных при объектной декомпозиции задачи?

2. Какие данные, характеризующие состояние объекта, необходимо представить в виде полей класса?

3. Какие методы, определяющие элементы поведения объектов класса, должен содержать каждый из классов?

4. Связаны ли между собой классы данной задачи? Если связаны, то как? Можно ли построить иерархию классов, стоит ли использовать композицию, наполнение или полиморфизм?

Для реализации объектов нашей программы необходимо разработать класс Символ, который должен включать поля, определяющие местоположение, размер и цвет символа, а также метод вывода символа.

Для реализации объектов типа Строка потребуется несколько классов: абстрактный базовый класс Строка, который будет содержать основные параметры строк и общие для всех типов строк методы инициализации Init и вывода OutS. Эти методы обращаются к пока неопределенным методам движения Move и связывания символов строки Tie, поэтому методы Tie и Move также объявлены в классе, но их реализация пока откладывается (абстрактные – «пустые» методы). Производные классы определяют конкретные виды

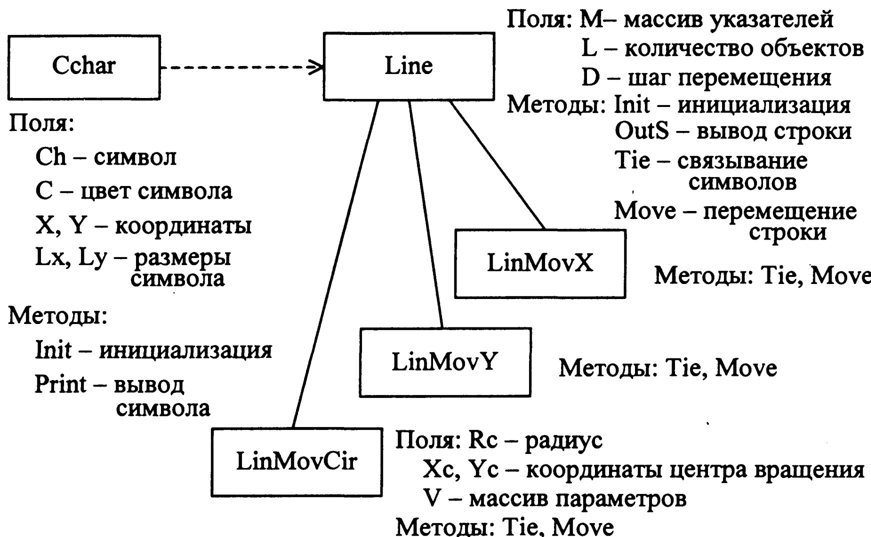


Рис. 2.11. Диаграмма классов программы «Текстовые эффекты»

движения: LinMovX – движение строки по горизонтали, LinMovY – движение строки по вертикали и LinMovCir – движение строки по окружности. Они будут наследовать поля и методы базового класса, перекрывать абстрактные методы Move и Tie и при необходимости добавлять собственные поля (рис.2.11).

Класс Line включает также поле M – массив указателей на объекты типа Cchar. Указатели используются исходя из того, что принципиально на базе класса Cchar можно построить другие более сложные классы, которые также могут управляться объектами класса Line и, соответственно, объектами классов, производных от Line. Из тех же соображений метод Print объявлен виртуальным. Таким образом, при разработке классов были применены специальные средства ООП: механизмы наследования, наполнения, полиморфизма.

Программу целесообразно разбить на два модуля: основную программу и библиотечный модуль, содержащий описание классов реализации предметной области задачи. При этом все внутренние поля и методы классов следует скрыть (объявить private).

На третьем этапе приступаем к созданию библиотеки классов и основной программы, т.е. уточняем типы полей, разрабатываем алгоритмы методов, тестируем и отлаживаем программу.

Ниже приведен текст модуля библиотеки:

```
Unit Ex_2_13a;
Interface
Type
 Pc=^Cchar;
```

```
 Cchar = Object              {класс Символ}
   private
     Ch:char;                {символ – основное поле}
     C,X,Y,
     Lx,Ly:integer; {цвет,координаты,размер символа}
   public
     Constructor Init(Chn:char;Cn,Xn,Yn:integer);
     Procedure  Print;virtual;  {вывод символа}
 End;
Type
 Line=object              {управляющий класс Линия}
     private
       M:array [1..100] of Pc;
       L:byte; {количество символов – объектов}
       D:real; {резервное поле – шаг перемещения}
     public
       Constructor Init(S:string;Cn,Xn,Yn:integer;Dn:real);
       Procedure OutS; {вывод строки}
     private
       Procedure Tie(i:byte;var X,Y:integer); virtual; {связывание символов}
       Procedure Move(i:byte);virtual; {перемещение}
End;
LinMovX=object(Line)
     private
          Procedure Tie(i:byte;var X,Y:integer); virtual;
          Procedure Move(i:byte);virtual;
 End;
LinMovY=object(Line)
     private
          Procedure Tie(i:byte;var X,Y:integer); virtual;
          Procedure Move(i:byte);virtual;
End;
LinMovCir=object(Line)
     private
          Rc,
          Xc,Yc:integer; {радиус, координаты центра вращения}
          V:array [1..100] of real; {угол смещения для каждого символа}
     public
          Constructor Init(S:string;Cn,Xcn,Ycn:integer;Dn:real);
     private
          Procedure Tie(i:byte;var X,Y:integer); virtual;
          Procedure Move(i:byte);virtual;
End;
```

```
Implementation
{********** тела методов класса Cchar *************}
Uses crt, graph;
Constructor Cchar.Init;
  Begin Ch:=Chn; {символ} C:=Cn; {цвет символа}
        X:=Xn; Y:=Yn;                    {начальные координаты}
        Lx:=Textwidth(Ch); Ly:=Textheight(Ch);   {размеры символа}
  End;
Procedure Cchar.Print; {вывод символа}
  Begin Setcolor(C); Outtextxy(X,Y,Ch)  End;
{********** тела методов класса Line  *************}
Constructor Line.Init;
Var i:integer;
 Begin
   L:=Length(S); D:=Dn;
   for i:=1 to L do
      begin  New(M[i],Init(S[i],Cn,Xn,Yn));
             Tie(i,Xn,Yn);
       end;
  End;
Procedure   Line.OutS;
  Var i,c:integer;
  Begin for i:=1 to L do begin
        c:=M[i]^.C; M[i]^.C:=Getbkcolor; M[i]^.Print;
        Move(i);    M[i]^.C:=c; M[i]^.Print end;
  End;
Procedure  Line.Tie; Begin  end;    {абстрактный метод}
Procedure  Line.Move; Begin End; {абстрактный метод}
{********** тела методов класса LinMovX ***********}
Procedure  LinMovX.Tie;
 Begin  X:=M[i]^.X+M[i]^.Lx+2; Y:=M[i]^.Y end;
Procedure  LinMovX.Move;
 Begin
   M[i]^.X:=Round(M[i]^.X+D);
   if (D>0) and (M[i]^.X>GetmaxX-M[i]^.Lx) then M[i]^.X:=0 ;
   if (D<0) and (M[i]^.X<0) then M[i]^.X:=GetmaxX-M[i]^.Lx;
 End;
{********** тела методов класса LinMovY ***********}
Procedure  LinMovY.Tie;
  Begin X:=M[i]^.X+M[i]^.Lx; Y:=M[i]^.Y-M[i]^.Ly   end;
Procedure  LinMovY.Move;
 Begin
   M[i]^.Y:=Round(M[i]^.Y+D);
   if (D>0) and (M[i]^.Y>Getmaxy-M[i]^.Ly) then M[i]^.Y:=0;
```

```
    if (D<0) and (M[i]^.Y<0) then M[i]^.Y:=Getmaxy-M[i]^.Ly;
  End;
{********** тела методов класса LinMovCir *********}
Constructor LinMovCir.Init;
 Var i,Xn,Yn:integer;
  Begin
    L:=Length(S); Rc:=Round((L+1)*2*Textheight(S)/(2*pi));
    D:=2*pi/(L+1); {шаг для исходной расстановки символов}
    V[1]:=pi; for i:=2 to L do V[i]:=V[i-1]+D;
    Xc:=Xcn; Yc:=Ycn;
    for i:=1 to L do
      begin Tie(i,Xn,Yn);
            New(M[i],Init(S[i],Cn,Xn,Yn));
      end;
    D:=Dn; {шаг перемещения}
  End;
Procedure LinMovCir.Tie;
  Begin X:=Round(Rc*cos(V[i])+Xc); Y:=Round(Rc*sin(V[i])+Yc) End;
Procedure LinMovCir.Move;
Begin
   V[i]:=V[i]+D;
   M[i]^.X:=Round(Rc*cos(V[i])+Xc);
   M[i]^.Y:=Round(Rc*sin(V[i])+Yc);
End;
End.
```

Текст основной программы при этом выглядит следующим образом:

```
Program Str_obj;
Uses Graph,Crt,Ex_2_13a; {модуль Ex_2_13a - библиотека классов}
Var Sx1,Sx2:LinMovX;  Sy1,Sy2:LinMovY;  Sc1,Sc2:LinMovCir;
    a,r:integer;    d:word;
Begin
  Clrscr;
  Write('Введите задержку: '); readln(d);
  a:=Detect; Initgraph(a,r,'C:\TP\BGI');
  Setbkcolor(8); Settextstyle(0,0,1);
  Sx1.Init('Перемещение слева направо',9,245,215,3);
  Sx2.Init('Перемещение справа налево',13,155,263,-3);
  Sy1.Init('Перемещение сверху вниз',11,165,250,2);
  Sy2.Init('Перемещение снизу вверх',12,300,400,-2);
  Sc1.Init('Вращение по часовой стрелке',15,
                              GetmaxX div 2,GetmaxY div 2,0.05);
```

```
  Sc2.Init('Вращение против часовой стрелки',
                              14,GetmaxX div 2,GetmaxY div 2,-0.05);
repeat    {циклический опрос методов перемещения объектов}
   Sx1.OutS; Sx2.OutS; Sy1.OutS;
   Sy2.OutS; Sc1.OutS; Sc2.OutS;
   Delay(d);
until Keypressed;
Closegraph;
End.
```

Вопросы для самоконтроля

1. Как определяется класс в Borland Pascal 7.0? Назовите основные компоненты класса. Где и как они описываются?

2. Как объявляются объекты класса? Перечислите способы инициализации полей. Для чего используется параметр Self? Когда его использование необходимо?

3. Перечислите основные виды отношений между классами. Какими средствами языка Borland Pascal 7.0 они реализуются? Приведите примеры использования отношений композиции и наполнения.

4. Какие виды полиморфизма реализованы в Borland Pascal 7.0? Определите, чем простой полиморфизм отличается от сложного. Перечислите три случая, когда использование сложного полиморфизма обязательно и объясните почему.

5. Какие объекты называются динамическими и в каких случаях они используются? Когда и как организуется контроль выделения памяти под размещение объекта и его полей?

6. Зачем создаются библиотеки объектов и какие средства для этого используются?

7. Назовите основные этапы разработки программных систем с использованием ООП. Определите задачи каждого этапа.

3. СРЕДСТВА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ В BORLAND C++ 3.1

Объектная модель С++, используемая Borland C++ 3.1, предоставляет программисту несколько большие возможности по сравнению с Borland Paskal 7.0. Так в языке реализованы более надежные средства ограничения доступа к внутренним компонентам класса, перегрузка операций, возможность создания шаблонов (как функций, так и классов).

3.1. Определение класса

В С++, так же как и в других языках программирования, *класс* – это структурный тип, используемый для описания некоторого множества объектов предметной области, имеющих общие свойства и поведение. Он описывается следующим образом:

```
class <имя класса>
{   private:    <внутренние (недоступные) компоненты класса>;
    protected:  <защищенные компоненты класса>;
    public:     <общие (доступные) компоненты класса>;
};
```

В качестве компонентов в описании класса фигурируют поля, используемые для хранения параметров объектов, и функции, описывающие правила взаимодействия с ними. В соответствии со стандартной терминологией ООП функции – компоненты класса или компонентные функции можно называть *методами*.

Компоненты класса, объявленные в секции *private*, называются внутренними. Они доступны только компонентным функциям того же класса и функциям, объявленным *дружественными* (раздел 3.5) описываемому классу.

Компоненты класса, объявленные в секции *protected*, называются защищенными. Они доступны компонентным функциям не только данного класса, но и его потомков. При отсутствии наследования – интерпретируются как внутренние.

Компоненты класса, объявленные в секции *public,* называются общими. Они доступны за пределами класса в любом месте программы. Именно в этой секции осуществляется объявление полей и методов интерфейсной части класса.

Примечание. Если при описании секции класса тип доступа к компонентам не указан, то по умолчанию принимается тип *private.*

Поля класса всегда описываются внутри класса. В отличие от полей, компонентные функции могут быть описаны как внутри, так и вне определения класса. В последнем случае определение класса должно содержать прототипы этих функций, а заголовок функции должен включать *описатель видимости*, который состоит из имени класса и знака «::». Таким способом компилятору сообщается, что определяемой функции доступны внутренние поля класса:

<тип функции> **<имя класса>::** <имя функции>(<список параметров>)
{<тело компонентной функции>}

По правилам С++, если тело компонентной функции размещено в описании класса, то эта функция по умолчанию считается встраиваемой (inline). Для таких функций компилятор помещает машинные коды, реализующие эту функцию, непосредственно в место вызова, что значительно ускоряет работу.

Так компонентные функции класса, описанного ниже, являются встраиваемыми по умолчанию:

```
#include <stdio.h>
#include <conio.h>
class X
  { private:     char c;
   public:      int x,y;
     /* определение встраиваемых компонентных функций внутри
                                   описания класса */
          void print(void)
             {  clrscr();
                gotoxy(x,y);    printf (" %c", c);
                x=x+10;   y=y+5;
                gotoxy(x,y);    printf (" %c", c);
             }
          void set_X(char ach,int ax,int ay) { c=ach;    x=ax;   y=ay; }
  };
```

Встраиваемую компонентную функцию можно описать и вне определения класса, добавив к заголовку функции описатель **inline.**

При определении компонентных функций следует иметь в виду, что не

разрешается объявлять встраиваемыми:

– функции, содержащие циклы, ассемблерные вставки или переключатели;

– рекурсивные функции;

– виртуальные функции.

Тела таких функций обязательно размещаются вне определения класса. Например:

```
class Y
  {           int x,y;
  public:     int k; char l;
             // прототипы компонентных функций
              void print(void);
              void set_Y(char al,int ax,int ay,int ak);
  };
// описание встраиваемой компонентной функции вне описания класса
inline void Y::set_Y(char al,int ax=40,int ay=15,int ak=15)
     {  x=ax;      y=ay;      k=ak;      l=al;      }
/* описание компонентной функции print() содержит цикл, значит
 использовать режим inline нельзя */
 void Y::print()
        { clrscr();
         gotoxy(x,y);
         for(int i=0;i<k;i++)  printf(" %c",l);     }
```

Рассмотрим пример определения класса, обеспечивающего работу со строками.

Пример 3.1. Определение класса (класс Строка). Пусть требуется описать класс, который обеспечит хранение и вывод строки:

```
#include <iostream.h>
#include <string.h>
class String   // начало описания класса
{  private:    char str[25];     // поле класса – строка из 25 символов
   public :
          // прототипы компонентных функций (методов)
          void set_str (char *);    // инициализация строки
          void display_str(void);  // вывод строки на экран
          char * return_str(void); // получение содержимого строки
};
//  описание компонентных функций вне класса
void String::set_str(char * s)    { strcpy(str,s);}
void String::display_str(void)   { cout << str << endl; }
char * String::return_str(void)  { return str; }
```

Определение класса может размещаться перед текстом программы, а может помещаться в отдельный файл, который подключается к программе при описании объектов этого класса с помощью директивы компилятора *include*:

– если файл находится в текущем каталоге – #include «имя файла»;

– если файл находится в каталогах автоматического поиска – #include <имя файла>.

В программе, использующей определенные ранее классы, по мере необходимости объявляются объекты классов. Такое объявление имеет вид:

<имя класса> <список объектов или указателей на объект>

Например:

```
String a, *b, c[6];
```

Приведенные переменные определяют три вида объекта класса string – объект *a*, указатель на объект *в* и массив из шести объектов *c*.

Как только объект класса определен, обращение к полям и функциям объекта может осуществляться с помощью полных имен, каждое из которых имеет вид:

<имя объекта>.<имя класса>::<имя поля или функции>.

Например:

```
a.String::str;
  b->String::set_str(st1);
  c[i].String::display_str().
```

Однако чаще доступ к компонентам объекта обеспечивается с помощью укороченного имени, в котором имя класса и двоеточие опускаются. В этом случае доступ к полям и методам осуществляется по аналогии с обращением к полям структур:

<имя объекта>.<имя поля или функции>;
<имя указателя на объект>-><имя поля или функции>;
<имя объекта>[индекс]. <имя поля или функции>.

Например:

```
a.str            b->str            c[i].str ;
a.display_str()  b->display_str()  c[i].display_str().
```

Первая строка демонстрирует обращение к полям простого объекта, объекта, описанного как указатель на объект, и элемента массива объектов. Вторая строка – обращение к методам соответствующих объектов.

Примечание. **Обращение из программы возможно только к общедоступным компонентам класса. Доступ к внутренним и защищенным полям и функциям возможен только из компонентных и «дружественных» функций.**

Так как объекты некоторого класса являются обычными переменными программы, на них распространяются общие правила длительности существования и области действия переменных. Поэтому при создании и уничтожении объектов определенного класса соблюдаются следующие правила:

- глобальные и локальные статические объекты создаются до вызова функции **main** и уничтожаются по завершении программы;
- автоматические объекты создаются каждый раз при их объявлении и уничтожаются при выходе из функции, в которой они появились;
- объект, память под который выделяется динамически, создается функцией **new** и уничтожается функцией **delete**.

Инициализация полей объектов. При объявлении полей класса не допускается их инициализация, поскольку в момент описания поля память для его размещения еще не выделена. Выделение памяти осуществляется не для класса, а для объектов этого класса, поэтому возможность инициализации полей появляется только после объявления объекта конкретного класса.

Значение может заноситься в поле объекта во время выполнения программы несколькими способами:

1) непосредственным присваиванием значения полю объекта;
2) внутри любой компонентной функции используемого класса;
3) согласно общим правилам C++ с использованием оператора инициализации.

Все вышеперечисленные способы применимы только для инициализации общедоступных полей, описанных в секции ***public***.

Инициализация полей, описанных в секциях ***private*** и ***protected***, возможна *только* с помощью *компонентной* функции.

Пример 3.2. Различные способы инициализации полей объекта

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class sstro
{ public:
   char str1[40];
   int x,y;
   void set_str(char *vs) // функция инициализации полей
```

```
    { strcpy(str1,vs);  x=0; y=0;}
  void print(void)  { cout<<' '<<x<<' '<<y<<' '<<str1<<endl;}
};
void main()
 {  sstro aa={"пример 1", 200, 400}; /* использование оператора
                               инициализации при создании объекта */
    sstro bb,cc;  // создание объектов с неинициализированными полями
    bb.x=200;  bb.y=150;   //инициализация общедоступных компонент
    strcpy(bb.str1, "пример 2"); //  при прямом обращении из программы
    cc.set_str("пример 3"); /* инициализация с помощью специальной
                               компонентной функции */
    aa.print(); bb.print(); cc.print();
 }
```

Однако использование указанных способов является не очень удобным и часто приводит к ошибкам. Существует способ инициализации объекта с помощью специальной функции, автоматически вызываемой в процессе объявления объекта (*конструктор* – раздел 3.2).

Значения полям объекта некоторого класса можно задать и с помощью операции присваивания ему значений полей другого, уже инициализированного объекта того же класса. В этом случае автоматически вызывается *копирующий конструктор* (раздел 3.2).

Неявный параметр this. Когда компонентная функция вызывается для конкретного объекта, этой функции автоматически и неявно передается в качестве параметра указатель на тот объект, для которого она вызывается. Этот указатель имеет специальное имя **this** и определен как константный в каждой функции класса:

<имя_класса> *const **this** = <адрес _объекта>.

В соответствии с описанием указатель изменять нельзя, однако в каждой принадлежащей классу функции он указывает именно на тот объект, для которого функция вызывается. Иными словами, указатель **this** является дополнительным (скрытым) параметром каждой нестатической (см. далее) компонентной функции. Этот параметр используется для доступа к полям *конкретного* объекта. Фактически обращение к тому или иному полю объекта или его методу выглядит следующим образом:

this->pole   this->str   this->fun().

Причем, при объявлении некоторого объекта A выполняется операция **this=&A**, а при объявлении указателя на объект b – операция **this=b**.

При работе с компонентами класса можно использовать этот указатель явным образом. Но следует отметить, что такое использование **this** не дает ощутимого преимущества, так как данные конкретных объектов уже доступны в принадлежащих классу функциях по именам.

Однако в некоторых случаях указатель **this** полезен и даже необходим. Очень удобным он становится, если в теле принадлежащей классу функции требуется явно задать адрес объекта, для которого она была вызвана.

Пример 3.3. Использование параметра this.

```
#include <iostream.h>
#include <conio.h>
class TA
 {  int x,y;
  public:
       void set (int ax,int ay){x=ax;y=ay;}
       void print(void)  { cout<<x<<' \t' <<y<<endl;}
      TA *fun1(){ x=y=100; return this; } /*функция возвращает указатель на
                                   объект, для которого вызывается */
      TA fun2(TA M) { x+=M.x; y+=M.y;  return *this;} /* функция возвращает
                              объект, для которого она вызывается */
 };
void main()
  {   clrscr();        TA aa,bb;
     aa.set(10,20);
     bb.fun1()->print();       // выводит: 100 100
     aa.print();               // выводит: 10 20
     aa.fun2(bb).print();       // выводит: 110  120
    getch();  }
```

Статические компоненты класса. Класс – это тип, а объект – конкретный представитель этого класса в программе. Для каждого объекта существует своя копия полей класса. Если все объекты одного типа используют некоторые данные совместно, то возникает проблема размещения этих данных и обеспечения их доступности из всех объектов класса. Решение возможно путем применения механизма с т а т и ч е с к и х к о м п о н е н т.

Статическими называются компоненты класса, объявленные с модификатором памяти **static**. Такие компоненты являются частью класса, но не включаются в объекты этого класса. Имеется ровно одна копия статических полей класса – общая для всех объектов данного класса, которая существует даже при отсутствии объектов данного класса.

Инициализация статических полей класса осуществляется только вне определения класса, но с указанием описателя видимости <имя класса>::. Например:

```
class point
 {    int x,y;
```

```
    static int obj_count;  /* статическое поле – (счетчик обращений)
                    инициализация в этом месте невозможна */
 public:
    point (){x=0;y=0; obj_count++;}  // обращение к статическому полю
 };
int point::obj_count=0;  // инициализация статического поля
```

Принципиально любой метод класса может обратиться к статическому полю и изменить его значение. Но существует возможность обращения к статическим полям класса при отсутствии объектов данного класса. Такой доступ осуществляется с помощью *статических компонентных функций* – компонентных функций, объявленных со спецификатором **static**.

Статические функции не ассоциируются с каким-либо объектом и не получают параметра **this**. Следовательно, они не могут без указания объекта обращаться к нестатическим полям класса. При необходимости ссылка на конкретный объект может быть передана в списке параметров, и тогда статическая функция может обратиться к нестатическим полям объекта следующим образом:

<имя объекта>.<имя нестатического поля класса>.

При обращении к статическим полям класса такой проблемы не возникает:

```
class point
 {  int x,y,color;       // нестатические поля
    static int obj_count; // статическое поле – счетчик обращений
 public:
   point ()
      {x=0;y=0;color=5;}
   static void draw_pount(point &p); //  статическая функция
 };
int point::obj_count=0;  //  инициализация статического поля
void point::draw_point(point &p) /* имя объекта передано в списке
                                                       параметров */
   { putpixel( p.x, p.y, p.color );// обращение к нестатическому полю
     obj_count++;}            // обращение к статическому полю
```

При обращении к статическим компонентам класса, являющимся принадлежностью всех объектов данного класса, можно вместо имени объекта указать имя класса:

<класс>::<компонент>.

Пример 3.4. Класс со статическими компонентами. В качестве примера, иллюстрирующего полезные свойства статических компонент, рассмотрим класс, использующий статические компоненты для построения списка своих объектов (рис. 3.1). В поле first хранится адрес первого элемента списка, в поле last – адрес последнего элемента списка. Нестатическое поле next хранит адрес следующего объекта. Полученный список затем используется для распечатки всех экземпляров класса с помощью статической функции drawAll(). Статическая функция drawAll() обращается к нестатическому полю next с указанием конкретного объекта

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include <alloc.h>
#include <stdio.h>
class String
  {  public:    char str[40];
     static String *first; //  статическое поле – указатель на начало списка
     static String *last; //  статическое поле – указатель на конец списка
     String *next;
     String(char  *s)
      { strcpy(str,s);
        next=NULL;
        if (first==NULL) first=this; else last->next=this;
        last=this; }
     void display()  { cout <<str<<endl; }
     static void displayA(); //  объявление статической функции
  };
String *String::first=NULL; //  инициализация статических компонент
String *String::last=NULL;
void String::displayAll () //  описание статической функции
   { String *p=first;
     if (p==NULL) return;
```

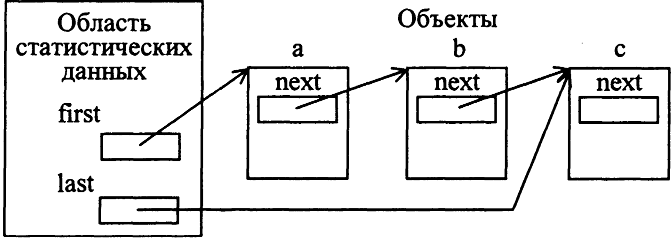


Рис. 3.1. Организация списка объектов для примера 3.4

```
      do { p->display(); p=p->next; }
      while(p!=NULL);
    }
int main(void)
{ String a("Это пример"),  //  объявление – создание трех объектов класса
        b("использования статических"),
        c("компонент");
  if (String::first!=NULL)  //  обращение к общему статическому полю
        String::displayAll(); //  обращение к статической функции
  getch();
}
```

Использование статических компонент класса позволяет снизить потребность в глобальных переменных, которые могли бы быть использованы с аналогичной целью.

Локальные и вложенные классы. Класс может быть объявлен внутри некоторой функции. Такой класс в С++ принято называть *локальным*. Функция, в которой объявлен локальный класс, не имеет непосредственного доступа к компонентам локального класса, доступ осуществляется с указанием имени объекта встроенного класса. Локальный класс не может иметь статических полей. Объект локального класса может быть создан только внутри функции, в области действия объявления класса. Все компонентные функции локального класса должны быть встраиваемыми.

Иногда возникает необходимость объявления одного класса внутри другого. Такой класс называется *вложенным*. Вложенный класс находится в области действия класса, внутри которого он объявлен. Соответственно, объекты этого класса могут использоваться как компоненты внешнего класса. Компонентные функции и статические компоненты вложенного класса могут быть описаны вне глобального класса.

Пример 3.5. Вложенные классы. Пусть требуется разработать программу, которая рисует ломаную линию или многоугольник отрезками линий, заданных координатами точек начала и конца отрезков.

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
 class Figura {
    class Point   // вложенный вспомогательный класс Точка
     {       int x,y,cv;
      public:
       int  getx(){return x;} // метод получения координаты x
       int  gety(){return y;} // метод получения координаты y
       int  getc(){return cv;} // метод получения цвета
      void setpoint(int ncv) // метод задания координат точки и ее цвета
```

```
        { cout<<" введите координаты точки:" <<endl;
          cin>>x>>y;      cv=ncv; }
    };
    class Line      // вложенный вспомогательный класс Линия
     {   Point Tn,Tk;  // начало и конец линии
       public:
        void draw(void)    // метод рисования линии
          { setcolor(Tn.getc());
          line(Tn.getx(),Tn.gety(),Tk.getx(),Tk.gety()); }
          void setline(int ncv) // метод задания координат точек линии
          { cout<<" введите координаты точек линии:" <<endl;
              Tn.setpoint(ncv);   Tk.setpoint(ncv);}
          void print()     // метод вывода полей класса
             { cout<<Tn.getx()<<'  ' <<Tn.gety()<<'  ';
               cout<<Tk.getx()<<'  ' <<Tk.gety()<<endl;    }
     };
  // поля и методы класса Фигура
  int kolline;        // количество отрезков Фигуры
  Line *mas;          // динамический массив объектов типа Линия
 public:
  void setfigura(int n,int ncv); // прототип метода инициализации объекта
  void draw(void);              // прототип метода отрисовки линий
  void delfigura(){delete [] mas;} // метод уничтожения объекта
  void print();                  // прототип метода вывода координат точек
 };
void Figura::setfigura(int n,int ncv)
 { kolline=n;
   cout<<" введите координаты" <<kolline<<" линий:" <<endl;
   mas=new Line [kolline];
   for(int i=0;i<kolline;i++){mas[i].setline(ncv);}  }
void Figura::draw(void)
 { for(int i=0;i<kolline;i++)  mas[i].draw();}
void Figura::print(void)
 { for(int i=0;i<kolline;i++)  mas[i].print(); }
void main()
{ int dr=DETECT, mod;  Figura Treangle;
  initgraph(&dr,&mod," c:\\borlandc\\bgi" );
  cout << " результаты работы: "<<endl; setbkcolor(15);  setcolor(0);
 Treangle.setfigura(3,6); // инициализировать поля объекта Треугольник
 Treangle.print(); // вывести значения координат точек
 getch();  setbkcolor(3);  cleardevice();
 Treangle.draw();       // нарисовать фигуру
 Treangle.delfigura();  // освободить динамическую память
 getch(); }
```

3.2. Конструкторы и деструкторы

Как упоминалось в предыдущем разделе, конструкторы и деструкторы представляют собой специальные методы класса, которые вызываются автоматически соответственно при создании и уничтожении объектов класса.

Конструкторы. По правилам C++ *конструктор* имеет то же имя, что и класс, может не иметь аргументов, никогда не возвращает значения и определяет операции, которые необходимо выполнить при создании объекта. Обычно это выделение памяти под поля объекта в динамической области и инициализация полей, но могут выполняться и другие операции.

Пример 3.6. Использование конструктора для инициализации полей класса. Рассмотрим использование конструктора для инициализации полей класса, описанного в примере 3.2, изменив доступность поля strl. Конструктор инициализирует поля класса sstr и осуществляет действия, связанные с проверкой длины вводимой строки и коррекцией строки, в случае несоответствия.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class sstr
 { private:      char str1[40];
   public:     int x,y;
     void print(void)
            { cout<<" содержимое общих полей: " << endl;
              cout<<"x= "<<x<<" y= "<<y<<endl;
              cout<<" содержимое скрытых полей: " << endl;
              cout<<" str1=" <<str1<<endl;}
     sstr(int vx,int vy,char *vs)  //конструктор класса sstr
      { int len=strlen(vs); // проверка правильности задания длины
        if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';} else strcpy(str1,vs);
        x=vx;y=vy;   }
};
void main()
{  clrscr();
   //создание объекта класса sstr при наличии конструктора
   sstr aa(200,150," пример использования конструктора ");
   aa.print();
   getch();
 }
```

Как и любая другая компонентная функция, конструктор может быть переопределяемым, указывая соответствующие действия для различных списков параметров. Поэтому класс может иметь несколько конструкторов,

позволяющих использовать разные способы инициализации объектов.

Пример 3.7. Использование переопределяемых конструкторов.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
 class sstr
 {private:     char str1[40];
  public:      int x,y;
     void print(void)
       { cout<<" содержание полей : " << endl;
         cout<<" x= " <<x<<" y= "<<y<<endl;
          cout<<" str1="<<str1<<endl;}
     sstr(void) // конструктор, определяющий поля по умолчанию
          { strcpy(str1,"конструктор по умолчанию" ); x=0;y=0; };
     sstr(char *vs) // конструктор, получающий значение поля str1
          {int len=strlen(vs);
           if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';} else strcpy(str1,vs);
           x=0;  y=0;   } // значения x и y задаются по умолчанию
      sstr(char *vs,int vx) // конструктор, получающий значения str1 и x
          {int len=strlen(vs);
           if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';} else strcpy(str1,vs);
           x=vx;  y=0;   } // значение y задается по умолчанию
     sstr(char *vs,int vx,int vy)  // конструктор, получающий все значения
         { int len=strlen(vs);
          if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';} else strcpy(str1,vs);
          x=vx;y=vy;    }
};
void main()
 {  clrscr();
    sstr a0, a1("пример конструктора 1 с x и y по умолчанию");
    sstr a2("пример конструктора 2 с y по умолчанию",100);
    sstr a3("пример использования конструктора 3",200,150);
    a0.print();
    a1.print();
    a2.print();
    a3.print();
    getch();
}
```

Кроме того, в самом конструкторе предусматривается возможность использования аргументов, заданных по умолчанию. В этом случае вместо нескольких переопределяемых конструкторов можно использовать один. Вернемся к предыдущему примеру и заменим все переопределяемые конструкторы одним.

Пример 3.8. Использование конструктора с аргументами по умолчанию.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class sstr
 { private:     char str1[40];
   public:
     int x,y;
     void print(void)
     { cout<<" содержимое полей :"<< endl;
       cout<<" x= " <<x<<" y= " <<y<<endl;
       cout<<" str1=" <<str1<<endl;}
     sstr(char *vs,int vx,int vy); /* прототип конструктора с аргументами по
                                       умолчанию */
 };
sstr::sstr(char vs="строка по умолчанию ",int vx=80,int vy=90) /* тело
                                       конструктора с аргументами по умолчанию */
   { int len=strlen(vs);
     if (len>=40) { strncpy(str1,vs,40);  str1[40]='\0'; }
     else strcpy(str1,vs);
     x=vx;y=vy;     }
void main()
 { clrscr();
   sstr a0,a1(" x и y по умолчанию" );
   sstr a2(" y по умолчанию " ,100);
   sstr a3("определяет  все поля",200,150);
   a0.print(); // выводит: x=80   y=90   str1= строка по умолчанию
   a1.print(); // выводит: x=80   y=90   str1= x и y по умолчанию
   a2.print(); // выводит: x=100  y=90   str1=  y по умолчанию
   a3.print(); //       выводит: x=200 y=150  str1=  определяет все поля
  getch();   }
```

В конструкторах может использоваться с п и с о к и н и ц и а л и з а ц и и, который отделяется от заголовка конструктора двоеточием и состоит из записей вида:

<имя> (<список выражений>),

где в качестве имени могут фигурировать:

– имя поля данного класса;

– имя объекта другого класса, включенного в данный класс;

– имя базового класса.

Список выражений определяет значения, используемые для инициализации соответствующих объектов.

Чаще всего конструктор со списком инициализации применяется для задания начальных значений объектам, в которых используются фиксированные и ссылочные поля, а также поля, являющиеся объектами других, ранее определенных классов.

Иногда возникает необходимость создать объект, не инициализируя его поля. Такой объект называется неинициализированным и под него только резервируется память. Для создания подобного объекта следует предусмотреть *неинициализирующий* конструктор. У такого конструктора нет списка параметров и отсутствует тело («пустой» конструктор). В этом случае при объявлении объекта с использованием такого конструктора значение полей объекта не определено.

Ниже приводится пример использования конструктора со списком инициализации и неинициализирующего конструктора для объектных и фиксированных полей.

Пример 3.9. Использование конструктора со списком инициализации и неинициализирующего конструктора.

```
#include <iostream.h>
#include <conio.h>
class integ
{    int num;
  public:
    void print(void)        { cout<<" num= "<<num<<endl;}
    integ(int v):num(v){} // конструктор со списком инициализации
    integ(void){}          // неинициализирующий конструктор
};
class obinteg
{    integ onum; // поле объектного типа
  public:
    const int x,y,c; // поля фиксированного типа
    void print(void)      { onum.print();
       cout<<" x=" <<x<<" y= "<<y<<" color="<<c<<endl;}
   obinteg(int va,int vx,int vy,int vc):onum(va),x(vx),y(vy),c(vc){} /*
      конструктор инициализирует поле объектного типа списком
      инициализации*/
   obinteg(void){}          // неинициализирующий конструктор
};
void main()
{ clrscr();
   integ aa(10); // инициализируемый объект
```

```
integ s1;        //s1 неинициализируемый объект
obinteg bb(20,40,100,13); // инициализируемый объект
obinteg s2; // неинициализируемый объект
cout<<"данные неинициализируемого объекта integ s1"<< endl;
s1.print();    // выводит случайные числа
cout<<"данные неинициализируемого объекта obinteg s2"<<endl;
s2.print();    // выводит случайные числа
cout << "данные  объекта класса integ: "<<endl;
aa.print();    // выводит заданные значения
cout << "данные объекта с объектным полем integ:" <<endl;
bb.print();    // выводит заданные значения
getch(); }
```

Не следует путать неинициализирующий конструктор с конструктором по умолчанию, так как последний иногда тоже не имеет списка параметров (пример 3.7), но в его теле задаются некоторые фиксированные значения полей. Если в определении класса уже описан конструктор по умолчанию (с параметрами или без таковых), то его можно использовать вместо неинициализирующего. При этом следует помнить, что поля объекта, образованного с помощью такого конструктора, будут определены. Использовать же в одном и том же классе оба вида конструкторов (конструктор по умолчанию и пустой конструктор) одновременно компилятор не позволяет.

Примечание. При использовании в программах массивов объектов некоторого класса наличие неинициализирующего конструктора обязательно, так как сначала создается массив объектов, а только затем каждому объекту индивидуально присваиваются начальные значения.

В С++ при объявлении объектов допускается использование операции присваивания, в правой части которой указано имя ранее определенного объекта. Например, для объектов класса, описанного в предыдущем примере, можно выполнить инициализацию следующим образом:

```
integ A(20,30,50,6), C=A;
```

В этом случае для инициализации полей объекта С активизируется специальный копирующий конструктор. *Копирующим* называется конструктор, который в списке параметров содержит параметр типа определяемого класса, например:

```
Class1 (const Class1&);
```

или

```
Class1(const Class1&,int=0);
```

Инициализация полей объекта при использовании копирующего конструктора выполняется методом копирования «поле за полем». При использовании данного метода последовательно выполняются конструкторы для всех полей инициализируемого объекта, причем в качестве параметров используются соответствующие значения полей объекта – параметра.

Копирующие конструкторы могут определяться в классе явно, а могут использоваться копирующие конструкторы, определенные по умолчанию. В последнем случае предполагается, что для каждого класса определен копирующий конструктор вида:

<имя класса> (**const** <имя класса>&),

за которым в зависимости от структуры класса может следовать список инициализации, включающий копирующие конструкторы базового класса и других полей. Этот конструктор в соответствии с предполагаемым описанием автоматически строится компилятором.

Примечание. Если для какого-либо поля или базового класса явно определен копирующий конструктор без **const**, то и конструктор класса в целом неявно определяется без **const**.

Пример 3.10. Использование предполагаемого копирующего конструктора.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class child
 {private: char *name;  int  age;
 public:
   void print(void)
       { cout<<" имя : "<<name;  cout<<" возраст: "<<age<< endl;}
   child(char *Name,int Age):name(Name),age(Age){}
 };
void main()
{ clrscr();
child aa("Мария",6),bb("Евгений",5);
cout << "результаты работы программы: "<<endl;
aa.print(); // выводит:   Мария и 6
bb.print(); //  выводит: Евгений и 5
child dd=aa; /*  предполагается использование копирующего конструктора */
dd.print();  // выводит: Мария и 6
  getch();
  }
```

В примере 3.10 по умолчанию предполагается конструктор:

*child(**const** child & obj):name(obj.name),age(obj.age){}*

В результате поля объекта **aa** без изменения копируются в поля объекта **dd.**

Если же необходимо при копировании полей изменять их содержимое, следует явно определить в описании класса копирующий конструктор, предусмотрев действия по изменению значений всех или некоторых полей.

Пример 3.11. Явное определение копирующего конструктора.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class intpole
 {   int cif;
  public:
  void print(void)    { cout<<"cif ="<<cif<<"  "<< endl;}
  intpole(int va)  {cif=va;}
  intpole(const intpole& ob1) // копирующий конструктор
         { cif= ob1.cif*3; }
 };
class obpole
 { int num;
      intpole intob; // объектное поле
 public:
   void print(void)
      { intob.print();
      cout<<" num= "<<num<<endl;}
  obpole(int vn,int va):num(vn),intob(va){}
 };
void main()
 {  clrscr();
    obpole aa(10,40);
   cout << " результаты работы программы: "<<endl;
   aa.print(); // выводится 40 и 10
   obpole bb=aa; // активизируется копирующий конструктор
   bb.print(); // выводится 120 и 10
   getch();  }
```

В приведенном примере для класса obpole предполагается копирующий конструктор по умолчанию:

obpole(const obpole &obj):num(obj.num) ,intob(obj.intob){ }

При инициализации объекта intob вызывается копирующий конструктор класса intpole, что приводит к инициализации соответствующего поля cif утроенным значением.

Кроме того, бывают случаи, когда поля объекта при копировании не изменяются, однако использование предполагаемого конструктора, автоматически создаваемого компилятором, может привести к непредсказуемым результатам.

Такая ситуация возникает, например, при передаче объекта в качестве параметра по значению некоторой функции. В этом случае, как известно, в памяти создается копия объекта, с которой должна работать функция. При создании копии должен вызываться копирующий конструктор, независимо от его наличия в описании класса. Автоматически генерируемый конструктор может не учесть некоторые особенности самого объекта и его полей. Особенно часто такие случаи происходят при работе с динамическими объектами и объектами, содержащими динамические поля (пример 3.30).

Деструкторы. Так же как конструктор используется для создания объекта, для его уничтожения используется специальная функция – *деструктор*. Имя функции совпадает с именем класса, в начале которого стоит символ «~» – (префикс тильда). Деструктор определяет операции, которые необходимо выполнить при уничтожении объекта. Обычно деструктор используется для освобождения памяти, выделенной под объект данного класса конструктором.

Деструктор не возвращает значения, не имеет параметров и не наследуется в *производных* классах (раздел 3.3). Класс может иметь только один деструктор или не иметь ни одного. Так как деструктор – это функция, то он может быть *виртуальным* (раздел 3.4). Однако отсутствие параметров не позволяет перегружать деструктор.

Деструктор вызывается неявно, автоматически, как только объект класса уничтожается. Момент вызова определяется моделью памяти, выбранной для объекта (локальный, глобальный, внешний и т.д.). Если программа завершается с использованием функции **exit,** то вызывается деструктор только глобальных объектов. При завершении программы, использующей объекты некоторого класса, функцией **abort** деструктор не вызывается. Однако, в отличие от конструктора, деструктор можно вызвать и явно.

Если в классе не объявлены конструктор и деструктор, то некоторые компиляторы автоматически проводят их построение. При этом конструктор используется для выделения памяти и размещения объекта после его описания, а деструктор – для корректного освобождения памяти после того, как имя объекта становится недействительным. В тех компиляторах, которые не проводят автоматического построения конструкторов и деструкторов при их отсутствии, выдается диагностическое сообщение, требующее явного объявления этих компонент класса.

Примечание. Если объект содержал указатель на некоторую динамически выделенную область памяти, то по умолчанию эта память не освобождается.

Пример 3.12. Определение деструктора в классе.

```
#include <iostream.h>
#include <string.h>
class din_string
{   char *str;    // поле класса – строка
    unsigned size;
public :
    din_string(char *s,unsigned SIZE=100); // прототип конструктора
   ~din_string();  // прототип деструктора
   void display(){cout <<str <<endl;}
};
din_string::din_string(char*s,unsigned SIZE) // конструктор
  { str = new char [size = SIZE];  strcpy(str,s);}
din_string::~din_string() //  описание деструктора вне класса
          { delete  [ ] str ;  }
void main()
{ din_string A ("my example");
  char ss [] = "ve cjvhuter is very good";
  din_string *p = new din_string(ss,sizeof(ss));
  p->display():
  delete p;        //  уничтожить объект – неявный вызов  деструктора
  A.display();}  // деструктор А будет вызван после окончания программы
```

3.3. Наследование

С++ поддерживает механизм наследования, являющийся одной из фундаментальных концепций ООП. Этот механизм обеспечивает создание производных классов. Производные классы – это эффективное средство расширения функциональных возможностей существующих классов без перепрограммирования этих классов и повторной компиляции существующих программ. В смысле функционирования производные классы или *потомки* являются более мощными по отношению к исходным (*базовым*) классам или *родителям*, так как, имея доступ ко многим функциям и полям базового класса, они обладают еще и своими компонентами.

По определению компонентами производного класса являются все компоненты базового класса, за исключением конструктора, деструктора и оператора равно (=), а также все компоненты, перечисленные в теле производного класса. Первое поле производного класса расположено после всех полей базового класса.

Ограничение доступа к полям и функциям базового класса (родителя) осуществляется с помощью специальных атрибутов, определяющих вид наследования.

Определение производного класса (потомка) выполняется следующим образом:

class <имя производного класса >:
    <вид наследования><имя базового класса>{<тело класса>};

где вид наследования определяется ключевыми словами: **private, protected, public.**

Видимость полей и функций базового класса из производного определяется видом наследования и представлена в табл. 3.1.

Таблица 3.1. **Видимость компонент базового класса в производном**

Вид наследования	Объявление компонентов в базовом классе	Видимость компонентов в производном классе
private	private	не доступны
	protected	private
	public	private
protected	private	не доступны
	protected	protected
	public	protected
public	private	не доступны
	protected	protected
	public	public

Если вид наследования явно не указан, то по умолчанию принимается **private**. Однако хороший стиль программирования требует, чтобы в любом случае вид наследования был задан явно.

Пример 3.13. Описание производного класса с типом доступа public.

```
#include <iostream.h>
#include <conio.h>
class A
 { private:   //защищенные (закрытые) компоненты класса
        int numza;
```

```
  public:     // общедоступные (открытые) компоненты класса
    int numoa;
    void print(void)
      { cout<<"защищенное поле A = "<<numza<<endl;
        cout<<"открытое поле A = "<<numoa<<endl;}
    A() { numza=20; numoa=50;}
  };
class B: public A   /* открытое наследование – классу B доступны все общие
                                             компоненты класса A */
{ private:   int numzb;
 public:
   void print(void)
      { cout<<"защищенное поле B = "<<numzb<<endl;
        cout<<"общее поле A =";
         cout<<numoa<<endl;}
   B(){numzb=100;}
};
void main()
  { clrscr();
    A aa; B bb;
    cout << "результаты работы: "<<endl;
    aa.print(); // выводит: защищенное поле A = 20  открытое поле A = 50
    bb.print(); // выводит: защищенное поле B = 100  общее поле A= 50
    getch();    }
```

Производный класс может по отдельности изменять доступность компонентов секций *protected* и *public* базового класса даже при самом жестком режиме наследования *private*. Однако в этом случае следует помнить, что описание доступности поля нужно помещать в ту же секцию, в которой оно описано в базовом классе.

Пример 3.14. Описание производного класса с видом наследования private.

```
#include <iostream.h>
#include <conio.h>
class A
{ private:    int numza;
  protected:   int numpra;
  public:
    int numoa;
    void print(void)
        { cout<<"закрытое поле класса A = "<<numza<<endl;
          cout<<"защищенное поле класса A= "<<numpra<<endl;
          cout<<"открытое поле класса A = "<<numoa<<endl;}
```

```
    A(){numza=20; numpra=30; numoa=50;}
};
class B: private A   // все компоненты класса не доступны в классе B
  { private:   int numzb;
   protected:
      A::numpra; /* защищенное поле класса A, объявляется доступным в
                                                   классе B */
   public:
     A::numoa; // общее поле класса A, объявляется доступным в классе B
     void print(void)
      { cout<<"закрытое поле класса B = "<<numzb<<endl;
        cout<<"открытое поле класса A,доступное в B=";
        cout<<numoa<<endl;
        cout<<"защищенное поле класса A,доступное в B = ";
        cout<<numpra<<endl;
      }
   B(){numzb=100;}
};
void main()
 { clrscr();
   A aa; B bb;
   cout << "результаты работы: "<<endl;
   aa.print(); // выводит:  20  30  50
   bb.print(); // выводит:  100  50  30
   getch();
 }
```

Производный класс может стать базовым для какого-либо другого класса. Множество связанных по наследованию классов образует иерархию классов.

Работа с объектом производного класса может осуществляться через указатель, объявленный для базового класса. Это возможно благодаря стандартному преобразованию указателя на производный класс в указатель на базовый, предусмотренному синтаксисом языка C++. Однако в этом случае указатель настроен на базовый класс и, как отмечалось в разделе 1.6, при работе с полями и методами производных классов возможны проблемы видимости этих компонент.

Конструкторы и деструкторы производных классов. Как было отмечено выше, в C++ конструкторы и деструкторы базового класса не наследуются в производных классах. Однако производный класс может иметь конструктор и деструктор, поэтому C++ поддерживает определенные правила взаимодействия между этими компонентами базовых и производных классов, которые необходимо соблюдать при использовании механизма наследования.

При объявлении объектов производного класса предусмотрен

автоматический вызов конструктора базового класса. Если конструктор базового класса имеет аргументы, то их значения должны быть переданы через дополнительные аргументы конструктора производного класса. При этом соблюдается строгий порядок конструирования объектов базового и производного классов: сначала создается объект базового класса и, возможно, его компоненты в порядке их объявления, а затем – объект производного класса.

Пример 3.15. Порядок работы конструкторов базового и производного классов.

```
#include <iostream.h>
#include <conio.h>
class A
  {    int a;
   public:
      int c1;
      void print(void)   { cout<<a<<"    "<< c1 <<endl;}
      A(int v):a(v){c1=10;} // конструктор базового класса А
};
class B: public A
 {    int b;
 public:
    void print(void)  { cout<<b<<"    "<<c1<<endl;}
   B(int va,int vb):A(va),b(vb){} // конструктор класса В
 };
class C: public B
 {    int c;
  public:
     void print(void) { cout<<c<<"    "<<c1<<endl;}
    C(int va,int vb,int vc):B(va,vb),c(vc){} // конструктор класса С
};
void main()
  { clrscr();
   A aa(10), *pa; // вызывается конструктор класса А
   B bb(10,100); //вызывается конструктор класса А, а затем – В
   C cc(10,100,1000); // вызываются конструкторы классов А, В и С
   cout << "результаты работы: "<<endl;
   bb.c1=25; // задание начальных значений общей компоненте c1
   cc.c1=35;
   aa.print();                    // выводит:  10      10
   bb.print();                    // выводит:  100     25
   cc.print();                    // выводит:  1000    35
```

```
 pa=&aa;
 pa->print();    // выводит:  10      10
 pa=&bb; /* указателю на объект базового класса pa присваивается указатель
            объект производного класса и осуществляется вызов
            компонентной функции */
 pa->print();              // выводит: 10      25
 pa=&cc;
 pa->print();              // выводит 10       35
 getch();
}
```

Отличие результатов печати при обращении к компонентной функции print() непосредственно с использованием полного имени объекта и при обращении через указатель объясняется тем, что указатель настроен на базовый класс и внутренние поля производных классов ему не доступны. По той же причине происходит вызов метода print() базового класса. Для получения правильного результата необходимо использовать механизм сложного полиморфизма (раздел 3.4).

Деструкторы, если они необходимы в программе, будут вызываться в порядке, обратном порядку вызова конструкторов. Если деструктор не определен, то он создается по умолчанию и при необходимости освобождает память, отведенную под динамический объект. Однако при использовании динамических полей, автоматического освобождения памяти не происходит, поэтому следует предусмотреть деструктор, который освобождал бы память, выделенную под динамические поля объекта используемого класса.

Пример 3.16. Последовательность описания и вызова конструкторов и деструкторов при многоуровневой иерархии классов.

Рассмотрим пример программы, реализующей иерархию классов, представленную на рис. 3.2.

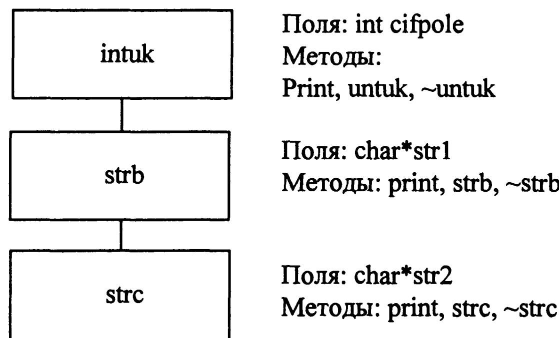


Рис. 3.2. Иерархия классов для примера 3.16

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class intuk
 {     int *cifpole;
  public:
    void print(void) { cout<<"cifpole ="<<*cifpole<<" "<< endl;}
    intuk(int va){cout<<"конструктор intuk"<<endl;
                    cifpole=new int;*cifpole=va;}
    ~intuk(){cout<< "деструктор intuk"<<endl; delete cifpole;} };
class strb: public intuk
 {    char *str1;
  public:
     void print(void)  { cout<<"str1= "<<str1<<endl;}
     strb(int va,char *vs):intuk(va)
       { cout<<"конструктор strb"<<endl;
         int l=strlen(vs); str1=new char [l+1]; strcpy(str1,vs); }
    ~strb(){ cout<< "деструктор strb "<< endl; delete str1;}
 };
class strc: public strb
 {      char *str2;
   public:
   void print(void)  { cout<<" str2="<<str2<<endl;}
   strc(int va,char *vs,char *vss):strb(va,vs)
      {cout<<"конструктор strc"<<endl;
       int k=strlen(vss); str2=new char [k+1];  strcpy(str2,vss);  }
  ~strc(){ cout<< "деструктор strc "<<endl;   delete str2;}
 };
void main()
   intuk aa(10); // выводит: конструктор intuk
   strb bb(10,"Объект 1  класса strb "); /* выводит: конструктор intuk
                                                     конструктор strb  */
   strc cc(10,"конструктор класса B", "объект 2  класса strc ");
          /* выводит: конструктор intuk
                      конструктор strb
                      конструктор strc */
   cout << " результаты работы программы: "<<endl;
   aa.print();     // выводит: cifpole =10
   bb.print();     // выводит: str1=  Объект 1  класса strb
   cc.print();     // выводит: str2=объект 2  класса strc
}   /* срабатывают деструкторы в порядке, обратном порядку  вызова
конструкторов: */
```

Множественное наследование. Язык С++ позволяет наследование не только от одного, а одновременно от нескольких классов. Такое наследование получило название *множественного.* При множественном наследовании производный класс включает произвольное количество базовых классов. Форма наследования в этом случае выглядит следующим образом:

class <имя производного класса>:
<вид наследования><имя базового класса 1>,
<вид наследования><имя базового класса 2>,
...
<вид наследования><имя базового класса n>{...};

Вид наследования определяет режим доступа к компонентам каждого из базовых классов. Базовые классы создаются в том порядке, в котором они перечислены в списке базовых классов при объявлении производного класса. Если конструкторы базовых классов не имеют аргументов, то производный класс может не иметь конструктора. *При наличии у конструктора базового класса одного или нескольких аргументов каждый производный класс должен иметь конструктор.* Чтобы передать аргументы в базовый класс, нужно определить их после объявления конструктора производного класса следующим образом:

<имя конструктора производного класса>(<список аргументов>):
<имя конструктора базового класса 1>(<список аргументов>),
......
<имя конструктора базового класса n>(<список аргументов>)
{<тело конструктора производного класса>}

Список аргументов, ассоциированный с базовым классом, может состоять из констант, глобальных параметров и (или) параметров для конструктора производного класса.

Последовательность активизации конструкторов такая же, как и для случая единственного базового класса: сначала активизируются конструкторы всех базовых классов (в порядке их перечисления в объявлении производного класса), затем конструкторы объектных полей и в конце конструктор производного класса.

Пример 3.17. Наследование от двух базовых классов.

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
class fixed
{   int numfx;
```

```
 public:
   fixed(int v):numfx(v)  {cout<<" вызов конструктора fixed\n";}
 };
class integral
{ public:
    int numfx;
    integral(int va):numfx(va)  {cout<<" вызов конструктора integ\n";}
 };
class rational
{ public:
    char c;int num;
    rational(int vn):num(vn) {cout<<" вызов конструктора rational\n";}
};
class twonasl:private integral, // наследование в защищенном режиме
                public rational // наследование в открытом режиме
{ private:    fixed numfix;    //  объектное поле
  public:
    twonasl(int nfx,int nm,char vc,int pnfx):
              integral(nfx),  rational(nm),  numfix(pnfx)
         {cout<<" вызов конструктора twonasl\n";
          c=vc;  // инициализация поля базового класса в теле
         }          // конструктора производного класса
      int get(void)    // функция получает значение внутреннего поля,
          {return numfx;}  //унаследованного от класса integral по private
};
void main()
 {clrscr();randomize;
   int r=random(100)-50;
   twonasl aa(r,20,'R',50);
   cout<<aa.get()<<' '<<aa.c<<' '<<aa.num<<endl;
}
```

Объект класса twonasl состоит из следующих полей:

– поля numfx, унаследованного от класса integral (описанного *public*, наследованного *private*, следовательно, *private*), инициализированного случайным числом в диапазоне от -50 до 49;

– полей num и *c*, унаследованных от класса rational (описанных *public*, наследованных *public*, следовательно, *public*), инициализированных числами 20 и символом «R», причем инициализация поля *c* в конструкторе класса rational не предусмотрена, поэтому она выполняется в теле конструктора класса twonasl;

– объекта класса fixed с именем numfix, включающего внутреннее поле numfx, не доступное в классе twonasl, поле объекта инициализируется числом 50.

В результате выполнения программы мы получаем идентификацию цепочки обращений к конструкторам:

вызов конструктора integ
вызов конструктора rational
вызов конструктора fixed
вызов конструктора twonasl

и содержимое полей numfx (класса integral), c и num, доступных в классе twonasl:

-49 R 20

Виртуальное наследование. При множественном наследовании базовый класс не может быть задан в производном классе более одного раза. Однако возможна ситуация, когда производный класс при наследовании от потомков одного базового класса многократно наследует одни и те же компоненты базового класса (рис. 3.3). Иными словами, производный класс будет содержать несколько копий одного базового класса.

Чтобы избежать многократного включения в производный класс компонент базового класса, используется *виртуальное наследование*. При виртуальном наследовании производный класс описывается следующим образом:

class<имя производного класса>:
virtual <вид наследования><имя базового класса>**{...};**

В этом случае включение в производный класс полей базового класса осуществляется один раз, а их инициализация происходит в производном классе, который не является прямым потомком базового класса. Вызов конструкторов при этом проводится в следующем порядке: сначала

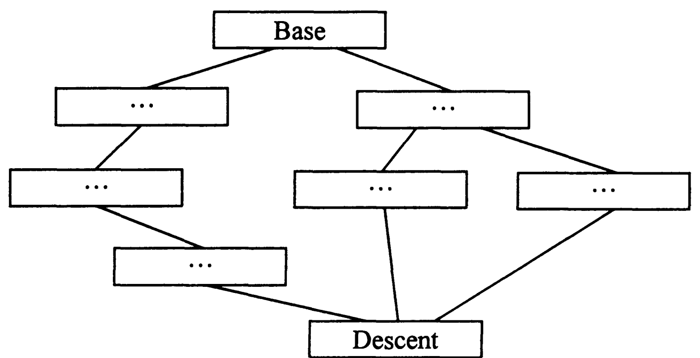


Рис. 3.3. Иерархия с многократным наследованием класса Base

конструктор виртуально наследуемого базового класса, затем конструкторы базовых классов в порядке их перечисления при объявлении производного класса, за ними – конструкторы объектных полей и конструктор производного класса. Деструкторы соответственно вызываются в обратном порядке.

Виртуально наследуемый класс обязательно должен содержать конструктор без параметров, который активизируется при выполнении конструкторов классов – прямых потомков виртуально наследуемого класса.

Пример 3.18. Виртуальное наследование.

```
#include <iostream.h>
class fixed
  { protected:   int Fix;
   public:
    fixed(void)           //  конструктор без параметров
      { cout<< "вызов конструктора fixed\n";}
    fixed(int  fix):Fix(fix)  //  конструктор с параметром
      { cout<<"вызов конструктора fixed int\n";}
  };
class derived_1:virtual public fixed // виртуальное наследование
  { public:
      int One;
     derived_1(void) { cout<<"вызов конструктора 1\n";}
  };
class derived_2:virtual private fixed  // виртуальное наследование
  { public:
      int Two;
    derived_2(void) { cout<<" вызов конструктора 2\n";}
  };
class derived: public derived_1,  /*  объявление производного класса –
                                         непрямого потомка */
                public derived_2
  { public:
    derived(void){ cout<<" вызов конструктора derived \n";}
    derived(int fix):
    fixed(fix)  { cout<<" вызов конструктора derived int \n";}
    void Out( )  {  cout<<" поле Fix ="<< Fix; }
  };
main()  { derived Var(10);  Var.Out( );  }
```

В результате работы программы получаем:

```
вызов конструктора fixed  int
вызов конструктора 1
вызов конструктора 2
вызов конструктора derived. int
поле Fix=10
```

В том случае, если бы наследование не было виртуальным, поле Fix было бы включено в объект класса derived дважды:

derived_1::Fix

и

derived_2::Fix.

3.4. Полиморфизм

Как было отмечено в разделе 1.5, для построения иерархий классов в любом языке программирования должен быть предусмотрен механизм полиморфизма, обеспечивающий возможность определения разных аспектов некоторого единого по названию метода для классов различных уровней иерархии. При этом различают *простой полиморфизм,* базирующийся на механизме раннего связывания, и *сложный полиморфизм,* использующий механизм позднего связывания.

Простой (статический) полиморфизм поддерживается языком C++ на этапе компиляции (раннее связывание) и реализуется с помощью механизма переопределения функций. Такие полиморфные функции называются в C++ *переопределяемыми*, при этом, в соответствии с общими правилами, они могут отличаться типом возвращаемого параметра и сигнатурой (количеством и типом передаваемых параметров).

Подобный подход позволяет строить более гибкие и совершенные иерархии классов, заменяя в производных классах методы в соответствии с требованиями разрабатываемой программы или системы, использующей эти классы. Для демонстрации использования статического полиморфного метода можно вернуться к примеру 3.15. В нем функция print() является статическим полиморфным методом. Пример можно модифицировать, введя в описание базового класса новую функцию show(), вызывающую метод print() и наследуемую в производных классах.

Пример 3.19. Использование раннего связывания.

```
#include <iostream.h>
#include <conio.h>
class A
 {   int a;
  public:
   void show(){cout<<" Содержимое полей объекта"<<endl;
                 print();}   //вызов статической полиморфной функции
  void print(void)     // первый аспект статической полиморфной функции
        { cout<<a<<endl;}
  A(int v):a(v){}
 };
```

```
class B: public A
 {  int b;
 public:
   void print(void) ) // второй аспект статической полиморфной функции
       { cout<<b<<endl;}
   B(int va,int vb):A(va),b(vb){}
 };
class C: public B
 {  int c;
 public:
    void print(void)  //третий аспект статической полиморфной функции
       { cout<<c<<endl;}
    C(int va,int vb,int vc):B(va,vb),c(vc){}
 };
void main()
{ clrscr();
  A aa(20);   B bb(10,100);   C cc(50,200,3000);
  cout << " Результаты работы: "<<endl;
  // явный вызов полиморфной функции print()
  aa.print();  // выводит: 20
  bb.print();  // выводит: 100
  cc.print();  // выводит: 3000
  getch();
 // неявный вызов полиморфной функции print()
 aa.show();   // выводит: 20
 bb.show();   // выводит: 10
 cc.show();   // выводит: 50
 getch();
 }
```

Нетрудно заметить, что результаты явного и неявного вызова метода print() различаются. Это объясняется тем, что метод show(), наследуемый в производных классах, вызывает метод print(). При раннем связывании show() жестко соединяется с этим методом на этапе компиляции. Поэтому, независимо от того, объект какого класса вызывает метод show(), из него будет вызван метод print() базового класса, для которого внутренние поля производных классов не доступны. Именно это является отличительной чертой раннего связывания.

Для получения правильного результата в подобных случаях необходимо использование сложного полиморфизма. Он реализуется механизмом позднего связывания и требует описания виртуальных функций.

Виртуальными называются функции, которые объявляются с использованием ключевого слова *virtual* в базовом классе и переопределяются

(замещаются) в одном или нескольких производных классах. При этом прототипы функций в разных классах должны совпадать не только по именам, но также по типу возвращаемого результата и сигнатуре, хотя алгоритмы, реализуемые такими функциями, как правило, отличаются между собой. Если типы функций различны, то механизм виртуальности для них не включается.

Примечание. Виртуальную функцию в С++ принято называть «полиморфной», что не совсем соответствует общепринятой терминологии, согласно которой она является «динамической полиморфной».

Как уже упоминалось выше, при использовании виртуальных функций нужный аспект полиморфной функции, вызываемой из метода базового класса, определяется на этапе выполнения, когда известно, для какого объекта вызван метод: объекта базового класса или объекта производного. Аналогично, если вызов функции осуществляется через указатель на базовый класс, нужный аспект функции вызывается в зависимости от того, адрес объекта какого класса будет содержать указатель.

В качестве иллюстрации можно несколько изменить предыдущий пример, добавив в его базовый класс ключевое слово *virtual* в определение функции print() и добавив в основную программу указатель на базовый класс.

Пример 3.20. Использование виртуальных функций.

```
#include <iostream.h>
#include <conio.h>
class A
{   int a;
   public:
  void show() {cout<<"Содержимое полей объекта"<<endl; print();}
  virtual void print(void) // описание виртуальности функции
             { cout<<a<<endl;}
    A(int v):a(v){} };
class B: public A
{  int b;
 public:
   void print(void) // первый аспект виртуальной функции
      { cout<<b<<endl;}
   B(int va,int vb):A(va),b(vb){}
};
class C: public B
{  int c;
 public:
   void print(void) // второй аспект виртуальной функции
      { cout<<c<<endl;}
```

```
    C(int va,int vb,int vc):B(va,vb),c(vc){}
 };
void main()
  { clrscr();
   A aa(10),*pa; // указатель на объект базового класса
   B bb(10,100);
   C cc(10,100,1000);
   cout << "     Результаты работы: "<<endl;
   cout << " Явный вызов полиморфной функции print(): "<<endl;
   aa.print(); // выводит: 10
   bb.print(); // выводит: 100
   cc.print(); // выводит:  1000
   getch();
   cout << " Неявный вызов полиморфной функции print(): "<<endl;
   aa.show();  // выводит: 10
   bb.show();  // выводит: 100
   cc.show();  // выводит: 1000
   getch();
   cout<<"Вызов  функции print() по указателю"<<endl;
   pa=&aa;    pa->print(); // выводит: 10
   pa=&bb;    pa->print(); // выводит: 100
   pa=&cc;    pa->print(); // выводит: 1000
   getch();
  }
```

Таким образом, каждый раз вызывается нужная версия (аспект) виртуальной функции.

Функция, объявленная виртуальной, остается таковой, сколько бы производных классов не образовывалось. Однако иногда в одном или нескольких производных классах переопределение виртуальной функции может отсутствовать. В этом случае механизм подключения виртуальной функции сохраняется, а, следовательно, вызывается функция класса, ближайшего к рассматриваемому. Отсутствие аспекта виртуальной функции не нарушает механизма позднего связывания, и если у наследников такого класса появится аспект виртуальной функции, он будет вызван без каких-либо нарушений.

Виртуальная функция обязательно должна быть компонентой некоторого класса. Она может быть объявлена дружественной другому классу, но не может быть объявлена статической (static).

Вызов виртуальной функции обычно реализуется как косвенный вызов по ТВМ (раздел 1.6). Эта таблица создается во время компиляции, а затем используется во время выполнения программы. Именно поэтому для вызова такой функции требуется больше времени.

Класс, который содержит хотя бы одну виртуальную функцию, называется *полиморфным*.

Абстрактные функции и классы. Существуют классы, которые выражают некоторую общую концепцию, отражающую основной интерфейс для использования в производных классах. Этот класс применяется при определении данных и методов, которые будут общими для различных производных классов. Он имеет смысл лишь как некоторая база. В этом случае смыслового определения виртуальной функции в базовом классе может не быть, но без ее описания корректная реализация сложной иерархии затруднена. Для решения этой проблемы в языке C++ используются абстрактные виртуальные функции.

Абстрактной виртуальной называется функция, объявленная в базовом классе как виртуальная, но не имеющая описания. Для описания абстрактной функции используется следующая форма:

virtual <тип><имя_функции>(<список параметров>)=0

Здесь «=0» – признак абстрактной виртуальной функции. При этом производный класс должен определить свою собственную версию функции, так как просто использовать версию, определенную в базовом классе, нельзя.

Класс, который содержит, по крайней мере, одну абстрактную виртуальную функцию, принято называть *абстрактным*. Создание объектов абстрактного класса запрещено. Этот класс может использоваться только как базовый для создания других классов.

Однако можно создавать указатель на объект абстрактного базового класса и применять его для реализации механизма полиморфизма. Кроме того, допускаются ссылки на абстрактные классы, позволяющие создавать временные объекты, для которых не требуется инициализации.

Если класс, производный от абстрактного базового класса, не содержит аспекта виртуальной функции, он автоматически становится тоже абстрактным.

Пример 3.21. Использование абстрактного класса при работе с полиморфными объектами. Пусть необходимо реализовать иерархию классов, представленную на рис. 3.4.

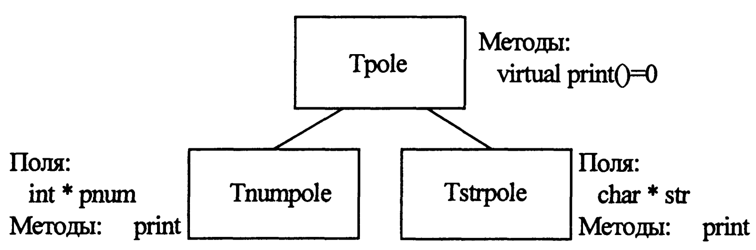


Рис. 3.4. Иерархия классов примера 3.21

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
 class Tpole  // абстрактный класс
  { public:
     virtual void Print(void)=0; // абстрактная виртуальная функция
     Tpole(){}
     ~Tpole(){}
  };
 class Tnumpole:public Tpole
  { private: int *pnum;
    public:
     Tnumpole(int n) {pnum=new int; *pnum=n; }
    ~Tnumpole(void){delete pnum;}
    void Print(void)  // аспект виртуальной функции
         {printf("\n число=  %d",*pnum);}
  };
class Tstrpole:public Tpole
 {    private: char *str;
   public :
     Tstrpole(char *st);
    ~Tstrpole(void){delete str;}
    void Print(void){printf("\n строка= %s",str);} /* аспект виртуальной
                                          функции класса Tstrpole*/
 };
 inline Tstrpole::Tstrpole(char *st)
    { int l;   l=strlen(st);   str=new char[l+1];   strcpy(str,st);  }
void main()
 { int n,i;
   char st[80];
   Tpole *a[10];    // массив указателей на базовый класс Tpole
  for(i=0;i<10;i++)
   { if (i%2)
        { printf("\nвведите целое число: "); scanf("%2d",&n);
          a[i]=new Tnumpole(n);} // Указатель на объект класса Tnumpole
    else
        {printf("\nвведите строку  < 80 байт: ");  scanf("%s",st);
         a[i]=new Tstrpole(st); } // Указатель на объект класса Tstrpole
   }
  puts("\n    РЕЗУЛЬТАТ ");
  for(i=0;i<10;i++)   { a[i]->Print();   delete a[i]; }
 }
```

Программа создает 10 полиморфных объектов, используя массив указателей на базовый класс Trole, который является абстрактным для данной иерархии классов. В процессе работы программы создаются объекты разных производных классов. Затем при печати содержимого массива осуществляется вызов нужного аспекта виртуальной функции.

3.5. Дружественные функции и классы

Как было отмечено ранее (раздел 3.1), механизм управления доступом позволяет выделить внутренние (private), защищенные (protected) и общедоступные (public) компоненты классов. Причем внутренние компоненты локализованы в классе и не доступны извне, а защищенные доступны только компонентным функциям класса и его наследникам. Такое ограничение на доступ к внутренним и защищенным компонентам класса может оказаться неоправданно строгим. Оно может существенно сужать возможности наследования других классов от данного и сокращать количество вариантов его использования.

Бывают случаи, когда не компонентные функции должны иметь возможность обращаться к внутренним компонентам класса. В такой ситуации класс может предоставить особые привилегии определенным внешним функциям или компонентным функциям другого класса. Эти функции получили название дружественных.

По определению, *дружественной функцией* класса называется функция, которая, не являясь компонентом некоторого класса, *имеет доступ ко всем его компонентам*. Функция не может стать другом класса «без его согласия». Для получения прав друга функция должна быть описана в теле класса со спецификатором **friend**. Именно при наличии такого описания класс предоставляет функции права доступа к защищенным и внутренним компонентам.

Пример 3.22. Внешняя дружественная функция.

```
#include <iostream.h>
#include <conio.h>
class FIXED
 {private:    int a;
  public: FIXED(int v):a(v){}
    friend void show(FIXED);
 };
void show(FIXED Par)
    { cout<<Par.a<<endl; // функция имеет доступ к внутреннему полю a
      Par.a+=10;
      cout<<Par.a<<endl;   }
```

```
void main()
  { clrscr();
    FIXED aa(25);
    cout << "результаты работы: "<<endl;
    show(aa); // выводит: 25   35
}
```

Друзьями класса могут являться и компонентные функции другого класса. Однако следует заметить, что такую привилегию методу другого класса может предоставить лишь автор данного класса, а не автор метода.

Пример 3.23. Дружественная функция – компонент другого класса.

```
#include <iostream.h>
#include <conio.h>
class FLOAT; // объявление класса без его определения
class FIXED
 {private:    int a;
  public:
    FIXED(int v):a(v){}
    double Add(FLOAT);
    void print(){cout <<a << endl;}
};
class FLOAT
 {private:   double b;
  public:
    FLOAT(double val){b=val;}
    void print(){cout <<b << endl;}
    friend double FIXED::Add(FLOAT); /* компонентная функция класса
                        FIXED объявляется дружественной классу FLOAT */
 };
double FIXED::Add(FLOAT Par)
   {return a+Par.b;}  /* функция получает доступ к внутреннему полю
                            класса FLOAT */
void main()
{ clrscr();
  FIXED aa(25); FLOAT bb(45);
 cout << "результаты работы: "<<endl;
 cout<<aa.Add(bb)<<endl; // выводит: 70
 aa.print();   // выводит: 25
 bb.print();   // выводит: 45
 getch();
}
```

Следует отметить некоторые особенности дружественных функций. Дружественная функция при вызове не получает указатель **this**. Объекты класса должны передаваться дружественной функции явно (через параметры).

Так как дружественная функция не является компонентом класса, на нее не распространяется действие спецификаторов доступа (public, protected, private). Место размещения прототипа дружественной функции внутри определения класса безразлично. Права доступа дружественной функции не изменяются и не зависят от спецификаторов доступа.

Использование механизма дружественных функций позволяет упростить интерфейс между классами. Например, дружественная функция может получить доступ к внутренним и защищенным компонентам сразу нескольких классов. Тогда из этих классов можно убрать компонентные функции, предназначенные только для обеспечения доступа к «скрытым» компонентам.

Поскольку ограничение доступа к дружественной функции не относится, работать она будет достаточно быстро, и написать ее не труднее, чем функцию доступа к обычной структуре С++.

Если необходимо, чтобы все функции некоторого класса имели доступ к внутренним полям другого класса, то весь класс может быть объявлен дружественным:

friend class <имя класса >.

Пример 3.24. Объявление дружественного класса.

```
#include <iostream.h>
#include <conio.h>
class SHOW; // объявление класса без его определения
class PAIR
{ private:    char *Head, *Tail;
  public:
    PAIR(char *one,char *two):Head(one),Tail(two){}
  friend class SHOW;  // объявление дружественного класса
};
class SHOW   /*всем функциям класса SHOW доступны внутренние поля
                              класса PAIR*/
{ private:  PAIR Twins;
  public:
    SHOW(char *one,char *two):Twins(one,two){}
   void Head(){cout <<Twins.PAIR::Head << endl;}
   void Tail(){cout <<Twins.PAIR::Tail << endl;}
};
void main()
{ clrscr();
  SHOW aa("HELLO","FRIEND");
```

```
  cout << "результаты работы: "<<endl;
  aa.Head(); // выводит: HELLO
  aa.Tail();   // выводит: FRIEND
 getch(); }
```

3.6. Переопределение операций

В С++ операции рассматриваются не как частично встроенные, а как модифицируемые элементы языка. Операции над стандартно определенными типами являются встроенными, т.е. программист не может повлиять на выполнение этих операций. Однако С++ предоставляет возможность определить любую из существующих операций для вновь созданных классов. Когда операция переопределена, ни одно из ее исходных значений не теряется. Просто вводится операция со схожим смыслом для объектов нового класса.

Переопределение операции реализовано как особый вид функции со специальным именем **operator@** (где @ – знак переопределяемой операции). Такие функции обычно называются функциями-операторами. Функция-оператор – может быть определена как внутри описания класса, так и вне его. Поэтому различают:

простую (определенную вне класса) функцию-оператор, которая бывает: одноместной (с одним аргументом) и двуместной (с двумя аргументами);

компонентную (определенную в классе) функцию-оператор, у которой первый аргумент всегда объект класса, задаваемый неявно и которая также бывает одноместной (не имеет явных аргументов) и двуместной (имеет один аргумент – второй операнд).

Общая форма определения функции-оператора приведена в табл. 3.2.

Таблица 3.2. **Формы описания функции-оператора**

Простая функция	Компонентная функция
Одноместная <тип результата>**operator** @ (аргумент)	Одноместная <тип результата>**operator** @ ()
Двуместная <тип результата>**operator** @ (<аргумент1>, <аргумент2>)	Двуместная <тип результата>**operator** @ (арг2)

В приведенных выше описаниях символ @ – любая допустимая операция, <тип результата> – тип возвращаемого значения (чаще всего того же типа, что и класс, хотя возможен и другой тип этого значения).

При переопределении операций следует помнить, что

– нельзя переопределять операции *,sizeof,?:,#,##,::,class::;

– операции =,[],() можно переопределять только в составе класса;

– переопределенная операция = не наследуется в производных классах;

– нельзя изменять приоритет и ассоциативность операции (порядок выполнения операций одного приоритета).

Функция-оператор допускает две формы вызова, которые представлены в табл. 3.3.

Таблица 3.3. **Формы вызова функции-оператора**

Стандартная форма	Операторная форма
Для простой функции **operator @** (<аргумент>) **operator @** (<аргумент1>,<аргумент2>)	Для простой функции @<аргумент> <аргумент1>@<аргумент2>
Для компонентной функции <аргумент>**.operator@()** <аргумент1>**.operator@** (<аргумент2>)	Для компонентной функции @<аргумент> <аргумент1>@<аргумент2>

При описании функции-оператора вне класса ей доступны только общие компоненты класса. Если необходимо, чтобы функция-оператор имела возможность обращаться к любым компонентам класса, то ее следует описать со спецификатором *friend*, определив ее дружественной этому классу.

Если необходимо, чтобы первый аргумент не был объектом класса, операция может переопределяться *только* вне класса. При переопределении операции вне описания класса хотя бы один аргумент должен быть объектом некоторого класса.

Пример 3.25. Описания функции-оператора вне класса.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class sstr// описание класса строка sstr
{ private: // скрытые поля класса
      char *str;
      int  len;
  public:  // общедоступные поля класса
    void print(void)
     { cout<<"Содержимое скрытых полей: "<< endl;
              cout<<" строка: "<<str<<endl;
               cout<<" длина строки :  "<< len<<endl;}
    sstr(){};
    sstr(char *vs)
      { len=strlen(vs);  str=new char[len+1];  strcpy(str,vs);  }
    ~sstr(){delete str;}
    friend  sstr & operator +(sstr &,sstr & ); /*переопределение операции «+» –
                              функция описывается вне класса */
};
sstr & operator +(sstr &A, sstr & B) // описание функции-оператора
```

```
  { int l =A.len +B.len;
    char *s;
    s=new char[l+1];    strcpy(s, A.str);  strcat(s,B.str);
    sstr *ps=new sstr(s);      delete []s;
    return *ps;   }
void main()
   sstr aa("пример использования ");
  sstr bb("переопределения операции");
  sstr cc=aa+bb+" добавка"; // операторная форма вызова
  cc.print();
  sstr dd;
  dd=operator +(aa,bb); // стандартная форма вызова
  dd.print();
  getch(); }
```

При описании функции-оператора внутри класса (как компонентной) первый аргумент *всегда* объект класса, что несколько ограничивает возможности ее использования.

Пример 3.26. Пример описания компонентной функции-оператора.

```
#include <string.h>
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
// описание класса строка sstr
class sstr
 { private: // скрытые поля класса
     char *str;
     int  len;
  public:
    void print(void)
       { cout<<"    содержимое скрытых полей: "<< endl;
         cout<<" строка: "<<str<<endl;
         cout<<" длина строки :  "<< len<<endl;}
    sstr(){}
    sstr(char *vs) { len=strlen(vs); str=new char[len+1]; strcpy(str,vs); }
    ~sstr(){delete str;}
   sstr & operator +(sstr &); /* прототип компонентной функции,
                                   реализующей бинарную операцию «+»*/
   sstr & operator -(); /* прототип компонентной функции, реализующей
                              унарную операцию «-» */
 };
```

```
sstr& sstr::operator -() /* описание переопределения  унарной  операции –
                                 обращение строки */
    { char c;
     for (int i=0;i<=len/2-1;i++){c=str[i];  str[i]=str[len-i-1]; str[len-i-1]=c;};
     return *this;      }
sstr & sstr::operator +(sstr &A) /* описание бинарной компонентной
                                        операции – функции */
   { int l =A.len+len;
     char *s;    s=new char[l+1];
     strcpy(s,str);     strcat(s, A.str);
     len=l; str=s;
     return *this; }
void main()
 { clrscr();
   sstr aa(" пример использования ");
   sstr bb("переопределения операции");
  sstr cc=aa+bb; // операторная форма вызова
  cc.print();
  sstr dd;   dd=aa.operator+(bb); // стандартная форма вызова
  dd.print();
  -aa;  -bb; // применение унарного минуса – обращение строк
  aa.print();      bb.print();
  getch(); }
```

При переопределении некоторых операций для обеспечения их коммутативности необходимо описывать несколько вариантов функций-операторов в теле класса.

Пример 3.27. Переопределение коммутативной операции «умножение на скаляр» и операции «+».

```
#include <iostream.h>
#include <conio.h>
class point
 {      float x,y;
  public:
   void print(void)
      { cout<<"   содержимое скрытых полей: "<< endl;
       cout<<" x ="<<x<<" y=  "<<y<<endl;}
  point(){}
  point(float ax,float ay):x(ax),y(ay){}
  point & operator +(point &p) // определение операции +
     { point *pp=new point;
      pp->x=x+p.x;     pp->y=y+p.y;
```

```
      return *pp;     }
  friend point & operator*(float k,point &p) /* определение коммутативной
                                              операции «умножение на скаляр» */
     { point *pp=new point;
       pp->x=p.x*k;     pp->y=p.y*k;
        return *pp;        }
  friend point & operator*(const point &p,float k) /* определение
                                        коммутативной операции «умножение на
                                        скаляр» */
    { point *pp=new point;
      pp->x=p.x*k;     pp->y=p.y*k;
      return *pp;     }
};
void main()
 { clrscr();
   point p(2,3),r(4,5),q;
   q=p.operator+(r);       // стандартная форма вызова
   q.print();              // выводит: 6 8
   q=p+r;                  // операторная форма вызова
   q.print();              // выводит: 6 8
   q=r+p;                  // операторная форма вызова
   q.print();              // выводит: 6 8
   q=operator*(5,p);       // стандартная форма вызова
   q.print();              // выводит: 10 15
   q=p*5;                  // операторная форма вызова
   q.print();              // выводит: 10 15
   q=5*p;
   q.print();              // выводит: 10 15
 }
```

При использовании в работе с классами библиотеки ввода–вывода C++ следует помнить, что классы ввода – вывода поддерживают операции «<<» (включение в поток) и «>>» (извлечение из потока) только для стандартных типов данных. Поэтому, как для определенных пользователем типов полей классов, так и для классов целиком при выводе объектов этих классов операции «<<» и «>>» следует переопределять для каждого класса.

При переопределении операций «включение в поток» и «извлечение из потока» используется следующая форма записи:

```
osrteam & operator<<(ostream & out,<новый тип> <имя>)
  {<тело функции-оператора>};
istream & operator >>(istream & in,<новый тип> &<имя>)
  {<тело функции-оператора>}.
```

Причем, при вовлечении в эти операции полей, описанных *private* и *protected*, операции «<<» и «>>» приходится описывать как дружественные.

Пример 3.28. Переопределение операций ввода–вывода.

```
#include <iostream.h>
#include <conio.h>
class TAdress
 { private:
     char  country[16];
     char  city[16];
     char  street[20];
     int  number_of_house;
   public:
     TAdress(){}
    ~TAdress(){}
    friend ostream& operator <<(ostream &out,TAdress obj);
    friend istream& operator >>(istream &in,TAdress &obj);
 };
 ostream& operator <<(ostream &out,TAdress obj) /* тело функции
                          переопределения операции  вставки в поток */
     { out<<" Адрес: "<<endl;
       out<<"  Country : "<<obj.country<<endl;
       out<<"  City    : "<<obj.city<<endl;
       out<<"  Street  : "<<obj.street<<endl;
       out<<"  House   : "<<obj.number_of_house<<endl;
       return out;      }
  istream& operator >>(istream &in,TAdress &obj) /* тело функции
                  переопределения операции извлечения  из потока */
     { cout<<" Введите адрес следующим образом:";
       cout<<" страна город улица  номер дома"; cout<<endl;
      in>>obj.country>>obj.city>>obj.street>>obj.number_of_house;
      return in; }
void main()
{  clrscr();
   TAdress a,b;
   cin>>a>>b;
    cout<<a<<b;
   getch();
 }
```

Операция присваивания, по умолчанию, определена для любого класса и обеспечивает покомпонентное присвоение, аналогичное тому, которое выполняется в копирующем конструкторе. При необходимости эту операцию также можно переопределить (пример 3.37).

В некоторых случаях такое переопределение обязательно, например, при работе с объектными полями.

3.7. Особенности работы с динамическими объектами

Как и для любой переменной языка C++, объекту некоторого класса память может быть выделена статически – на этапе компиляции или динамически – во время выполнения программы.

При статическом распределении память под объект выделяется и освобождается в соответствии с классом памяти используемой переменной (автоматическая, внешняя, внешняя статическая и т.д.). Причем известно, что в отличие от других языков и даже С, C++ предоставляет возможность определять переменную не только до описания основной программы, но и в любом месте основной программы (иногда это называют динамической инициализацией). Но даже в этом случае память под такой объект выделяется на этапе компиляции из объема памяти, отведенной для программы, а инициализация полей происходит во время ее выполнения.

При динамическом распределении память под объект класса выделяется на этапе выполнения из так называемой *свободной* (динамической) памяти. Причем реальное выделение и освобождение участков осуществляется в момент, определяемый программистом (при выполнении функций new и delete) и полностью им контролируется.

Однако даже при статическом распределении памяти под сам объект память под отдельные его поля может выделяться динамически. Поэтому в зависимости от того, как выделяется память под сам объект и его поля, различают:

– статические объекты с динамическими полями;

– динамические объекты со статическими полями;

– динамические объекты с динамическими полями.

Определение классов для реализации каждого из названных видов объектов имеет некоторые особенности в языке C++, связанные со спецификой его использования.

Статические объекты с динамическими полями. В C++ можно выделить динамически память под переменную любого типа, определенного к этому моменту.

Для этого предусмотрено несколько способов, но наиболее удобным является использование функции **new.** Удобство механизма, предоставляемого **new**, заключается в том, что для его применения не требуется определять конкретный размер выделяемой памяти, как в других случаях. Необходимый размер определяется автоматически типом переменной, хотя количество таких участков можно указать. Операция **new** выделяет требуемую память для размещения переменной и возвращает адрес выделенного участка, преобразуя указатель к нужному типу. Если по каким-либо причинам память не может быть выделена, функция возвращает нулевой указатель (NULL). Освобождается память, выделенная по **new**, с помощью функции **delete** также в соответствии с типом переменной.

Однако динамически выделять память под простые переменные невыгодно, так как память расходуется не только под саму переменную, но и под информацию о выделении памяти. Поэтому динамическое распределение памяти целесообразно использовать при работе со сложными структурами данных, особенно, если размер будущего поля заранее не известен (например, определяется в процессе вычислений или зависит от входной информации).

Наиболее часто динамическое распределение памяти применяют в классах, использующих в качестве полей массивы, строки, структуры и их комбинации. В этом случае поле содержит *указатель* на переменную соответствующего типа. Указатель получает свое значение при выделении памяти под структуру данных, для которой он определен. Освобождение этой памяти происходит по адресу в указателе в соответствии с типом переменной.

Конструктор такого класса обычно осуществляет выделение участков памяти требуемого размера под переменные, адреса которых присваиваются соответствующим указателям, а также обеспечивает контроль наличия доступной памяти. Деструктор класса в этом случае должен при уничтожении объекта освобождать распределенную при конструировании память, так как по умолчанию эта память не освобождается.

Пример 3.29. Конструирование и разрушение объектов с динамическими полями.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class Tmass
 { public:
    int *mas; // поле класса – динамический массив целых чисел
    int size;  // размер массива
   void print(void);
   Tmass(){};
   Tmass(int Size);
   ~Tmass(){delete []mas; // освобождение выделенной памяти
        cout<<"Деструктор"<<endl;}
};
Tmass::Tmass(int Size)
   { cout<<"Конструктор"<<endl;
     mas=new int[Size]; // выделение памяти под массив
     size=Size;
     cout<<"Введите " << size;
     cout << " значений массива"<<endl;
     for(int i=0;i<size;i++)   cin>>mas[i];     }
void Tmass::print(void)
```

```
      { cout<<"   содержимое массива: "<< endl;
       for(int i=0;i<size;i++)  cout<<mas[i]<<" ";
        cout<<endl;}
void main()
  { clrscr();
   Tmass aa(6),cc(4);  // два раза вызывается конструктор класса
    aa.print();  cc.print();
   getch();
 }  // вызывается деструктор для двух объектов
```

При переопределении операций ввода–вывода конкретные значения полей объекта становятся известными только после ввода данных. Поэтому для более эффективного и корректного использования памяти целесообразно реальное распределение памяти под поля такого объекта выполнять не с помощью конструктора класса, а во время операции ввода. Конструктор в этом случае должен быть неинициализирующим.

Следует помнить, что возможен ввод новых значений полей в уже существующий объект (например, многократное использование некоторого объекта в качестве промежуточной переменной). Тогда, прежде чем выделять память под новое значение поля, необходимо освободить память, выделенную ранее под предшествующее значение, так как автоматическое освобождение неиспользуемой памяти, выделенной по ***new***, компилятором не предусмотрено. В противном случае эти участки останутся в занятом состоянии после замены содержимого указателя на адрес новой области.

При инициализации объекта значениями полей другого объекта, как отмечалось в разделе 3.2, используется копирующий конструктор. Кроме того, если объект является параметром некоторой функции, передаваемым по значению, внутри этой функции создается копия такого объекта, при построении которой также применяется копирующий конструктор. Если копирующий конструктор в определении класса не предусмотрен, он создается по умолчанию. При использовании динамических полей, результат действия такого конструктора не предсказуем, так как компилятор не может точно определить размеры копируемых полей. В этом случае пользователь должен предусмотреть собственный копирующий конструктор, обеспечивающий грамотное выполнение действий, предусмотренных при копировании.

Пример 3.30. Использование собственного копирующего конструктора.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class TAdress
```

```
{ private:
    char * country;
    char * city;
    char * street;
    int  number_of_house;
  public:
    TAdress(TAdress &a) // копирующий конструктор
      { cout<<" Копирующий конструктор"<<endl;
        country=new char[strlen(a.country)+1];
        city=new char[strlen(a.city)+1];
        street=new char[strlen(a.street)+1];
        strcpy(country,a.country);
        strcpy(city,a.city);
        strcpy(street,a.street);
        number_of_house=a.number_of_house;   }
   TAdress() {cout<<" Конструктор класса"<<endl;
             country=NULL; city=NULL; street=NULL; }
   ~TAdress()
       { cout<<" Деструктор класса"<<endl;
         delete []  country;  delete [] city;  delete [] street;  }

   TAdress& operator =(TAdress &obj) /* переопределение операции
                                      присваивания */
    { if(country!=NULL) delete [] country;
      country=new char[strlen(obj.country)+1]; strcpy(country,obj.country);
      if(city!=NULL) delete [] city;
      city=new char[strlen(obj.city)+1];  strcpy(city,obj.city);
      if(street!=NULL) delete [] street;
      street=new char[strlen(obj.street)+1];  strcpy(street,obj.street);
      number_of_house=obj.number_of_house;
      return *this;  }
   friend ostream& operator <<(ostream &out,TAdress obj);
   friend istream& operator >>(istream &in,TAdress &obj);
};
ostream& operator <<(ostream &out,TAdress obj)
  {out<<" Адрес: "<<endl;
   out<<" Country : "<<obj.country<<endl;
   out<<" City    : "<<obj.city<<endl;
   out<<" Street  : "<<obj.street<<endl;
   out<<" House   : "<<obj.number_of_house<<endl;
  return out;    }
 istream& operator >>(istream &in,TAdress &obj)
  { char str[40];   int l;
```

```
    cout<<" Введите адрес в порядке:";
    cout<<" страна, город, улица, номер дома";  cout<<endl;
    in>>str;     l=strlen(str);
    if(obj.country!=NULL)   // если память выделена,
            delete [] obj.country; // то – освободить ее
    obj.country=new char[l+1];   strcpy(obj.country,str);
    in>>str;     l=strlen(str);
    if(obj.city!=NULL)    // если память выделена,
            delete [] obj.city;  // то – освободить ее
    obj.city=new char[l+1];  strcpy(obj.city,str);
    in>>str;     l=strlen(str);
    if(obj.street!=NULL) // если память выделена,
            delete [] obj.street; // то – освободить ее
    obj.street=new char[l+1];  strcpy(obj.street,str);
    in>>obj.number_of_house;
    return in;     }
void main()
{ clrscr();
  TAdress a,b,c;
  cin>>a>>b;
  cout<<a<<b;
  getch();
  c=a;   // выделяется память и происходит копирование
  cout<<c;
  a=b;  // уничтожается ранее выделенная память, выделяется новая, затем
                    происходит копирование */
  cout<<a;
  getch();
}
```

Следует также отметить особенности реализации оператора присваивания. Как уже отмечалось, операция присваивания значений полей одного объекта другому предусмотрена по умолчанию. Однако использование динамических полей объекта приводит к не совсем корректному ее выполнению. Операция присваивания осуществляется, как и при работе копирующего конструктора, поле за полем. Поскольку реальным содержимым полей являются указатели, то происходит замена старого адреса на новый. При этом память, ранее адресуемая указателем, останется не освобожденной, так как адрес ее будет потерян. Поэтому рекомендуется в описании класса предусмотреть переопределение операции присваивания с учетом всех особенностей работы с выделением и освобождением памяти для полей определяемого класса (см. пример 3.30).

Динамические объекты со статическими полями. Как и на другие типы данных, на объект любого класса можно ссылаться через указатель. При этом значение ему можно присвоить с помощью операции взятия адреса (в этом случае он может адресовать и статический объект). Однако более эффективно использовать указатели для реализации механизма динамического распределения памяти под объект из свободной области.

Поскольку типом объекта является некоторый класс, то применительно к объектам используется следующая форма обращения к функции New:

<имя_указателя_на_объект> = **new** <имя_класса>;

или

<имя_указателя_на_объект>=**new**<имя_класса>(<список_параметров>);

Вторая форма используется при наличии у конструктора списка параметров.

Функция delete требует указания только имени объекта:

delete <имя указателя на объект>;

Пример 3.31. Использование простых динамических объектов.

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
class TVector
 { private:   int x,y,z;
  public:
   TVector(){cout<<"пустой конструктор"<<endl;}
   TVector(int ax,int ay,int az)
       {cout<<" конструктор"<<endl; x=ax;y=ay;z=az;}
   ~TVector(){cout<<" деструктор"<<endl;}
    void PrintVec();
 };
 void TVector::PrintVec()
    {cout<<"значение вектора: "<<setw(5)<<x<<" , ";
      cout<<setw(5)<<y<<" , "<<setw(5)<<z<<"\n";   }
void main()
{ TVector *a,*b; // определяются два указателя на объекты класса
  clrscr();
  // выделяется память под динамические объекты класса
  a=new TVector(12,34,23); // вызывается конструктор
  b=new TVector(10,45,56); // вызывается конструктор
```

```
  a->PrintVec(); // выводит: 12, 34, 23
  b->PrintVec(); // выводит: 10, 45, 56
// освобождается память, выделенная под динамические объекты класса
  delete a; // вызывает деструктор
  delete b; // вызывает деструктор
 }
```

Если используется массив динамических объектов, то выделять память под него можно:

– либо одним непрерывным фрагментом (равным объему всех объектов массива) в момент его определения в программе, например:

B mas[] = *new B*[*n*];

– либо в цикле под каждый конкретный элемент массива индивидуально, например:

for (*i*=0; *i*<*n*; *i*++) *mas*[*i*]=*new B*;

Освобождать выделенную память нужно так, как она была выделена: одним фрагментом или поэлементно

delete[] *mas*;

или

for (*i*=0; *i*<*n*; *i*++)*delete mas*[*i*];

Кроме того, в C++ массив элементов некоторого класса разрешено определять только, если объект класса может быть объявлен без указания инициализирующего значения. Следовательно, требуется предусмотреть в описании класса неинициализирующий конструктор или конструктор по умолчанию. В первом случае поля объектов элементов массива будут не определены, а во втором – окажутся инициализированными одинаковыми значениями. Если программисту совершенно необходимо проинициализировать элементы массива различными значениями, можно написать конструктор по умолчанию, который непосредственно или косвенно считывает и записывает нелокальные данные, или предусмотреть специальный метод инициализации объектов.

Пример 3.32. Обработка массива динамических объектов. Рассмотрим программу, создающую массив из пяти динамических объектов в виде массива указателей на эти объекты и массив из трех объектов через указатель на первый объект массива.

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
class sstr
 {private:   char str1[40];
   public:
     int x,y;
     void print(void)
       { cout<<"    содержимое полей : "<< endl;
         cout<<" x= "<<x<<" y= "<<y<<"  str1= "<<str1<<endl;}
     sstr(){cout<<"неинициализирующия конструктор"<<endl;}
     sstr(int vx,int vy,char *vs); // прототип конструктора по умолчанию
    ~sstr(){cout<<" деструктор "<<endl;}
    void setstr(int ax,int ay,char *vs); //  функция определения полей объекта
 };
sstr::sstr(int vx,int vy,char *vs=" конструктор со строкой по умолчанию ")
   { int len=strlen(vs);
     if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';}  else strcpy(str1,vs);
    x=vx;  y=vy;     cout<<" конструктор по умолчанию"<<endl;     }
 void  sstr::setstr(int ax,int ay,char *vs)
   { int len=strlen(vs);
     if (len>=40) {strncpy(str1,vs,40);str1[40]='\0';}  else strcpy(str1,vs);
      x=ax;  y=ay;   }
void main()
{ clrscr();
  sstr   *a[5], // массив указателей на пяти динамических объектов типа sstr
          *c;    // указатель на массив динамических объектов
 char *vs="sstraaffgghhjj";       /*выделить память и инициализировать объект
                                            */
 for(int i=0;i<5;i++) a[i]=new sstr(10+i,10+2*i,"aaaaaaaa"+i); /* создать
                          массив из пяти динамических объектов */
 for(i=0;i<5;i++) a[i]->print();// вывести содержимое полей объектов
 for(i=0;i<5;i++) delete a[i]; // освободить память
 c=new sstr[3];          // выделить память под  три динамических объекта
 for(i=0;i<3;i++     ) //  инициализировать поля динамических объектов
    {(c+i)->setstr(15+i,12+i*2,vs+i);;}
 for(i=0;i<3;i++) (c+i)->print();// вывести содержимое полей объектов
 delete []c; // освободить память
 getch();
}
```

Необходимо отметить также особенности работы с динамическими объектами классов, входящих в некоторую иерархию. Как уже отмечалось, C++ позволяет присваивать указателям на базовый класс, значение указателей на любой из производных от него. В этом случае возникают проблемы с доступом к полям объекта, описанным в производном классе:

– указатель на объект базового класса связан с описанием его полей, и поля, описанные в производном классе для него «невидимыми» (рис 1.25). Поэтому при обращении через указатель на базовый класс к полям производного объекта необходимо средствами языка явно переопределить («привести») тип указателя;

– кроме того, если при определении указателя на базовый класс создается динамический объект производного класса, то во время уничтожения такого объекта вызывается деструктор лишь базового класса и память освобождается не корректно, так как деструктор не может правильно определить размеры освобождаемой памяти. Эта проблема решается применением *виртуального деструктора*. Если при объявлении деструктора базового класса описать его как *virtual*, то все конструкторы производного класса также будут виртуальными. При уничтожении объекта с помощью оператора **delete** через указатель на базовый класс будут корректно вызваны деструкторы всех производных классов. Все эти особенности имеют место и при работе с полиморфными объектами (раздел 3.4)

Пример 3.33. Использование указателей на базовый класс и виртуального деструктора. В программе определены два класса: класс, содержащий поле целого типа и производный от него класс, содержащий в качестве поля динамический массив, размер которого определяется значением поля целого типа, унаследованного от родителя.

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
class integ
 { protected: int n;
  public:
     virtual void print(void) { cout<<"  "<<n<<endl;}
     integ(int vn){cout<<"конструктор integ"<<endl;  n=vn; }
     virtual ~integ() { cout<<"деструктор integ"<<endl; }
 };
class masinteg: public integ
 {  int *mas;
 public:
   masinteg(int vn);
  ~masinteg(){ delete [] mas;  cout<<"деструктор masinteg"<<endl;  }
   void print(void);
```

```
};
masinteg::masinteg(int vn):integ(vn)
  { cout<<"конструктор masinteg"<<endl;
    cout<<"создается "<<n<<" элементов массива"<<endl;
    mas=new int[n];    for(int i=0;i<n;i++)  mas[i]=random(30); }
void masinteg::print()
  { cout<<" содержимое массива " <<endl;
    for(int i=0;i<n;i++) {cout<<"  "<<mas[i];   cout<< endl; } }
void main()
 { clrscr(); randomize();
  integ *pa;
  cout << "результаты работы: "<<endl;
  pa=new integ(5);
  pa->print();
  delete pa;    // вызывается деструктор integ
  pa=new masinteg(6);
  pa->print();
  delete pa;   // вызывается деструктор masinteg
  getch();
 }
```

Необходимо еще раз отметить особенности выполнения операции присваивания при работе с объектами через указатели. Дело в том, что присваивание одного объекта другому с помощью указателей сводится к переадресации этого указателя: после выполнения операции первый указатель содержит тот же адрес, что и второй – адрес второго объекта. Старое значение адреса (адрес первого объекта) стирается, и память, выделенная ранее под первый объект, остается не освобожденной. С другой стороны, при попытке уничтожить объекты по указателям возникает ошибка, так как один и тот же участок памяти, выделенный под второй объект, освобождается дважды. Во избежание этих ошибок необходимо учитывать и контролировать подобные ситуации (пример 3.37).

На класс, используемый для динамических объектов, можно возложить задачу управления памятью, если переопределить operator new() и operator delete(). Особенно это полезно для классов, которые являются базовыми для многочисленных производных классов. Примеры подобного переопределения подробно рассмотрены в [2].

Динамические объекты с динамическими полями. Такие объекты сочетают характерные особенности статических объектов с динамическими полями и динамических объектов.

Все вышеприведенные примеры предполагают наличие безграничной или достаточно большой динамической (свободной) памяти. Однако реально эта память конечна, и при большом количестве объектов и значительном количестве

памяти под их поля может наступить ситуация, когда память либо под сам объект, либо под его поля не может быть выделена. В таком случае в конструкторе или функции, выполняющей распределение, следует предусмотреть контроль наличия памяти. При обнаружении недостатка памяти необходимо определить, в какой момент возникла ситуация (при распределении памяти под объект или под поле), а затем грамотно выйти из создавшегося положения.

3.8. Параметризованные классы

В C++ предусмотрена реализация еще одного типа полиморфизма – параметризованные классы и параметризованные функции.

Параметризованный класс. Это некий шаблон **(template)**, на основе которого можно строить другие классы. Этот шаблон следует рассматривать как описание множества классов, моделирующих абстрактную структуру данных и отличающихся типами полей, включенных в эту структуру. Хорошим примером параметризированных классов могут служить шаблоны реализации списков, массивов, множеств и т.п. Шаблон классов определяет правила построения каждого отдельного класса из множества разрешенных (допустимых) классов. Описание шаблона выглядит так:

template <список параметров шаблона> <описание класса>

В списке параметров шаблона могут присутствовать как параметры, определяющие тип, так и параметры, для которых этот тип фиксирован. Каждый формальный параметр, определяющий тип, обозначается ключевым словом **class.**

Пример 3.34. Шаблон, позволяющий формировать одномерные динамические массивы из заданных элементов.

```
template < class type>  // объявление шаблона с аргументом «type»
class array    // начало описания класса с именем array
{ type * contents;  // указатель на динамический массив типа type
  int size;           //  размер массива
 public:
   array(int number) {contents = new type [size=number];}
  ~array (){delete [] contents;}
  type & operator [] (int x)  // переопределение операции  []
   { if ((x<0)||(x>=size))  { cerr << "ошибочный индекс"; x=0;}
      return contents[x];
   }
}  // конец описания класса с именем array
```

Используя шаблон, можно определить объекты класса array с аргументом любого допустимого типа. Допустимыми являются как встроенные типы языка, так и типы, определенные пользователем.

Формат определения объекта одного из классов, порождаемых шаблоном, выглядит следующим образом:

имя_параметризованного_класса <список_параметров_шаблона>
имя_объекта (параметры_конструктора)

Например, для шаблона классов, описанного в примере 3.34, можно определить следующие объекты:

```
array <int>  int_a(50);  // объект – массив с элементами целого типа
array <char>  char_a(100); // объект – массив с элементами типа char
array <float> float_a(30);  // объект – массив с элементами типа float
```

Описание шаблона можно записать в файл с расширением «*h*» и, используя его стандартным образом, определять объекты заданного множества классов, а также выполнять операции над этими объектами.

Пример 3.35. Использование шаблонов для формирования массивов и печати их элементов.

```
#include "array.h"
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
void main()
 { int i;
   array<int> int_a(5);
   array<char>  char_a(5);
   for (i=0;i<5;i++)     //  определение компонент массивов
        {int_a[i]=random(20);  char_a[i]:='A'+i;}
   puts("компоненты массива");
  for (i=0;i<5;i++)
        {cout << "   "<< int_a[i]<< "   " <<char_a[i]<<endl;}
}
```

Функции параметризованного класса можно определить и вне описания класса, но следует учитывать, что функции являются параметризованными и их надо описывать как функции – шаблоны (*параметризованные функции*), например:

```
template <class type> type & array < type>::operator[](int x) /* заголовок функции
                                                   – шаблона*/
  { if ((x<0)||(x>=size))  { cerr << "ошибочный индекс";x=0;}
   return contents[x];     }
```

Параметризованная функция. Помимо шаблона классов разрешается определять также шаблоны функций. *Шаблон функции* – это глобальная функция, определенная за пределами классов. В отличие от перегрузки функции, при которой для каждой сигнатуры создается своя функция, шаблон семейства функций определяется один раз, но это описание содержит параметры. Для задания параметров используется список.

Описание шаблона функции:

template <список параметров шаблона> <описание функции>

Каждый формальный параметр обозначается ключевым словом **class**, за которым следует имя параметра (идентификатор). Например, описание шаблона функции, возвращающей значение максимальной из двух переменных, выглядит следующим образом:

```
template <class type> type   max(type x, type y){ return(x>y)?x:y;}
```

Использование шаблона функций позволяет передать в функцию в качестве параметра тип используемых в ней данных, а далее выполнять операции, предусмотренные алгоритмом над объектами заданных типов. Если для некоторых типов объектов операции, используемые в функции, не определены, следует ввести явное описание функции для этого типа. Например, при использовании шаблона из предыдущего описания, если в качестве аргумента будут использованы строки, то, так как операция «>» для строк не определена, функция выдаст неправильный результат. Для того чтобы в качестве параметра шаблона можно было использовать строки, следует добавить явное описание функции-оператора «>» для строк.

Пример 3.36. Использование шаблонов функций при создании шаблонов классов.

```
#include <conio.h>
#include <string.h>
#include <iostream.h>
template <class T> T   max(T x, T y)
   { return(x>y)?x:y;}
char * max(char * x, char * y)  /* описание функции для объектов типа
                                    строка */
{return strcmp(x,y) > 0? x:y;}
void main ()
{  clrscr();
   int a=1,b=2;      char c='a', d='m';     float e=123.675,f=456;
  char str[]="abcd", str2[]="abnd";
  //вызов функции для объектов различного типа
```

```
  cout << "Integer max=  "<<max(a,b)<< endl;  // Integer max=   2
  cout << "Character max= "<<max(c,d)<< endl; // Character max= m
  cout << "Float max=    "<<max(e,f)<< endl;     // Float max=    456
  cout << "String max=   "<<max(str,str2)<< endl; // String max=  abnd
  getch();
}
```

Примечание. Каждый аргумент шаблона функций должен определять тип как минимум одного аргумента описываемой функции. Это гарантирует, что нужный вариант функции будет выбран на основе анализа типов ее аргументов. Отмеченное правило не распространяется на шаблоны классов.

Рассмотрим обобщенный пример использования шаблонов классов при моделировании структуры типа «множество».

Пример 3.37. Использование шаблона классов (шаблон классов «Множество»)

Пусть необходимо разработать шаблон классов для реализации объектов типа множество. Шаблон должен обеспечивать возможность выполнения следующих операций: построения множества из заданных элементов, добавления элементов, удаления элементов, пересечения множеств, объединения множеств, присвоения множеств и проверки вхождения элемента во множество.

Для корректной работы с множеством доступ к его элементам осуществляется с помощью внутренних функций. Чтобы присвоение одного объекта другому происходило с освобождением памяти, выделенной под значения полей объекта приемника, предусмотрена переопределенная операция присваивания. Типовые операции над множествами реализованы несколькими способами: через дружественные внешние операции-функции, компонентные операции и в виде обычных компонентных функций. Так как возможна инициализация объекта другим объектом-множеством, предусмотрен копирующий конструктор.

Шаблон описывает правила выполнения всех указанных выше операций относительно параметра type (рис. 3.5).

Примечание. Этот пример можно рассматривать как иллюстрацию использования объектов с динамическими полями, учитывающий некоторые особенности таких объектов.

Тестирующая программа объявляет объект класса Множество с элементами типа char и проверяет выполнение всех операций.

```
#include <iostream.h>
#include <conio.h>
```

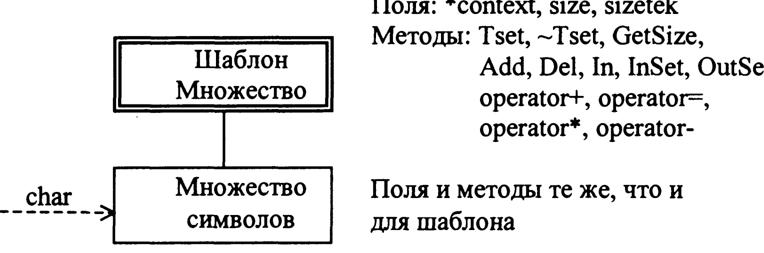


Рис. 3.5. Структура классов для примера 3.37

```
#include <string.h>
template < class type>   // объявление шаблона с аргументом type
class Tset
{ private:
   type *contents;  // указатель на динамический массив типа type
   int size;         //  размер множества при инициализации
   int sizetek;      //текущий размер множества
   int SetEl(int ind,type m) // функция записи элемента в массив
     {  if (ind<size)   { contents[ind]=m;sizetek+=1;return 1;}
        else {cout<<"размер множества превышен"<<endl; return 0;}  }
   type GetEl(int ind)   // функция чтения элемента из массива
     { return contents[ind];}
public:
   Tset(){ cout<<" конструктор по умолчанию "<<endl; contents=NULL;}
   Tset(Tset &A);        // прототип копирующего конструктора
   Tset(int number)      // конструктор класса Tset
     { cout<<" конструктор"<<endl;
       contents = new type [size=number];sizetek=0;}
   ~Tset ()              // деструктор
       { cout<<" деструктор"<<endl; delete [] contents;}
    int  Getsize()   //определение максимального размера множества
       {return size;}
    void Add(type m);  // добавление элемента к множеству
    void Del(type m);  // удаление элемента из множества
    int  In(type m);     // проверка вхождения элемента во множество
    int  InSet(int Nn);    // заполнение множества
    void  OutSet();        // вывод элементов множества
    Tset& CrossSet(Tset &B); // пересечение множеств
    Tset &operator +(Tset &B); // объединение множеств
    Tset &operator =(Tset &B); // присвоение множеств
    friend Tset &operator *(Tset &A,Tset &B); // пересечение множеств
    friend Tset &operator -(Tset &A,Tset &B); // дополнение  множеств
};
```

```
template <class type> Tset<type>::Tset(Tset<type>&A)
   { cout<<" Копирующий конструктор"<<endl;
     contents = new type [size=A.Getsize()];
     sizetek=A.sizetek;
     for(int i=0;i<sizetek;i++)   contents[i]=A.contents[i];   }
template <class type> int Tset<type>::InSet(int Nn)
   { type c; int k=0;
     if (Nn>Getsize()) Nn=Getsize();
     cout<<"Введите "<<Nn<<" элементов множества: "<<endl;
     for (int i=0;i<Nn;i++)
          { cin>>c;  if (!In(c)){if(!SetEl(k,c)) return 0;k+=1;}}
     return 1;     }
template <class type> void Tset<type>::OutSet()
   { cout<<" Содержимое множества: "<<endl;
     if (sizetek!=0)  {for(int i=0;i<sizetek;i++) cout<<"  "<<GetEl(i);}
     else cout<<" Пустое множество";
     cout<<endl;   }
template <class type> int Tset<type>::In(type m)
   { for (int i=0;i<sizetek;i++) if (m==GetEl(i)) return 1;
     return 0;     }
template <class type> void Tset<type>::Add(type m)
   {if (!In(m))   if (sizetek<size) SetEl(sizetek,m);   }
template <class type> void Tset<type>::Del(type m)
   { int h;
     if (In(m))  { h=0;
                   for(int i=0;i<sizetek;i++)
                          if(h) contents[i-1]=contents[i];
                          else if (m==GetEl(i)) h=1;
                   sizetek-=1;    }
   }
template <class type> Tset<type>
  & Tset<type>::operator =(Tset<type> &B)
   { cout<<" Операция присваивания "<<endl;
     if (this==&B) return *this;
     if (contents!=NULL) delete [] contents;
     contents = new type [size=B.Getsize()];
     sizetek=B.sizetek;
     for(int i=0;i<sizetek;i++)  {contents[i]=B.contents[i];}
     return *this;     }
template <class type> Tset<type>
  & Tset<type>::operator +(Tset<type> &B)
   { for(int i=0;i<B.sizetek;i++) Add(B.GetEl(i)); return *this;  }
template <class type> Tset<type>
  &Tset<type>::CrossSet(Tset<type> &B)
   { int i=0;
```

```
      do  {if( !B.In(GetEl(i))) Del(GetEl(i)); else i++;}
      while (i<sizetek);
      return *this;     }
template <class type> Tset<type>
   & operator -(Tset<type> &A,Tset<type> &B)
    { Tset<type> *C=new Tset<type>(A.Getsize());
      for(int i=0;i<A.sizetek;i++) if(!B.In(A.GetEl(i))) C->Add(A.GetEl(i));
      return *C;    }
template <class type> Tset<type>
  & operator *(Tset<type>&A,Tset<type> &B)
     { int l;
       if(A.Getsize() > B.Getsize()) l=A.Getsize();  else l=B.Getsize();
       Tset<type> *C=new Tset<type>(l);
       for(int i=0;i<A.sizetek;i++)
         {if( B.In(A.GetEl(i))) C->Add(A.GetEl(i));}
       return *C;   }
void main()
{int n; clrscr();
Tset<char> aa(15),bb(10),dd(10),cc;
cout<<"Введите число членов формируемого  множества n<= ";
cout<<aa.Getsize()<<endl;   cin>>n;  aa.InSet(n);
cout<<"Введите число членов формируемого  множества n<= ";
cout<<bb.Getsize()<<endl;   cin>>n; bb.InSet(n);
cout<<"Введите число членов формируемого  множества n<= ";
cout<<dd.Getsize()<<endl;   cin>>n;  dd.InSet(n);
clrscr();
cout<<" Множество aa "<<endl;  aa.OutSet();
cout<<" Множество bb "<<endl;  bb.OutSet();
cout<<" Множество dd  "<<endl; dd.OutSet();
Tset<char> ee=aa*bb; // инициализировать объект уже созданным
cout<<" Пересечение множеств aa и bb"<<endl; ee.OutSet();
aa.CrossSet(bb);
cout<<" Пересечение множеств aa  и  bb компонентное"<<endl;
aa.OutSet();
aa=aa+dd;
cout<<" Объединение множеств aa и dd"<<endl;  aa.OutSet();
cc=dd-bb;
cout<<" Дополнение множеств dd и bb"<<endl;   cc.OutSet();
cc=dd;  // явная операция присвоения
cc.OutSet();
}
```

Тестирующая программа осуществляет формирование трех множеств букв и операции над ними, а также фиксирует последовательность и место вызова конструкторов различного назначения и деструкторов.

3.9. Контейнеры

Решение многих задач подразумевает создание наборов объектов в различных формах и обработку таких наборов.

Объект, назначением которого является хранение объектов других типов и управление ими, принято называть *контейнером*. Классическими примерами контейнеров являются списки, вектора, ассоциативные массивы. Иногда содержимое контейнеров называют *последовательностями*.

В С++ контейнеры можно реализовать с использованием контейнерных и параметризованных классов.

Контейнерные классы. Данный термин обозначает распространенный прием разработки классов, использующий механизмы композиции или наполнения для подключения некоторых объектов к управляющему объекту – контейнеру. Контейнерный класс содержит в своем определении несколько объектных полей или полей – указателей на объекты. Если контейнерный класс использует механизм композиции, то тип и количество управляемых объектов жестко определены типом и количеством объектных полей. Если он использует механизм наполнения, то подключение реализуется через указатели, следовательно, контейнер может управлять как объектами некоторого базового, так и объектами всех потомков этого класса.

Обычно контейнерные классы реализуют некоторые типовые структуры, такие как массив, стек или список и типовые операции над данными, которые могут быть записаны в эти структуры или прочитаны из них.

Основная операция любого контейнерного класса – последовательный просмотр объектов. Такая обработка обеспечивается двумя способами.

Первый способ базируется на создании специальной процедуры просмотра всех элементов контейнера. В эту процедуру в качестве параметра передается имя функции или процедуры, реализующей алгоритм требуемой обработки элемента контейнера.

Пример 3.38. Контейнерный класс с процедурой поэлементной обработки. Пусть требуется разработать контейнер на базе сортированного списка элементов, принадлежащих иерархии классов Число – Строка – Таблица.

Структуру классов будем разрабатывать поэтапно. В основу иерархии классов положим классы Список – Элемент. Класс Список будет содержать три поля – указатели на элементы – объекты класса Элемент: указатель на первый элемент, указатель на последний элемент и указатель на текущий элемент. Эти указатели используются для организации двусвязного списка. Класс Элемент содержит только два поля – указатели на элементы того же класса, которые будут хранить адрес следующего элемента и адрес

предыдущего элемента списка. Эти два класса образуют контейнерный класс, управляемые объекты которого должны наследоваться от класса Элемент (рис. 3.6).

Затем на базе этих классов разработаем абстрактный контейнерный класс Сортированный список, предусматривающий метод сортировки Sort с внутренним вызовом метода сравнения элементов Compare. Наличие внутреннего метода сравнения позволит в дальнейшем на базе этого класса создавать другие классы, использующие различные законы сортировки.

От класса Элемент наследуем классы предметной области задачи: Число, Строка, Таблица, добавляя новые поля и перекрывая метод печати содержимого элемента Print.

Теперь можно описать класс Пользовательский сортированный список, который перекроет метод сравнения элементов при сортировке, задав сравнение реальных полей классов предметной области задачи.

Ниже представлена программа тестирующая разработанную иерархию классов:

```
#include <stdio.h>
#include <conio.h>
 class TElement      // абстрактный класс Элемент списка
   { public:
        TElement *pre, *suc; /* Два поля – ссылка на предыдущий и ссылка на
                                   последующий элементы */
        virtual void Print(void)=0;     // абстрактная функция печати
        TElement(void) { pre=suc=NULL;}      // конструктор
        virtual ~TElement(void)          // деструктор
            { puts("Уничтожить элемент.");}
   };
 class TSpisok      // класс Список
   { protected: TElement *first, *last, *cur; /* поля – указатели соответственно
                       на первый, последний и текущий элементы списка */
```


Рис. 3.6. Иерархия классов для примера 3.38

```
    public:
        void Add(TElement *e);          // добавление элемента в список
        TElement *Del(void);            // удаление элемента из списка
        void ForEach(void (*f)(TElement *e)); /*выполнить для каждого
                                                  элемента */
        TSpisok(void){ first=last=cur=NULL;}
        virtual ~TSpisok(){ puts("Уничтожить список.");}
  };
 void TSpisok::Add(TElement *e)
    { if (first==NULL)  first=last=e;
      else { e->suc=first; first=first->pre=e;}     }
TElement *TSpisok::Del(void)
    { TElement *temp;
      temp=last;
      if (last!=NULL) { last=last->pre;  if (last!=NULL) last->suc=NULL; }
      if (last==NULL) first=NULL;
      return temp;      }
void TSpisok::ForEach(void (*f)(TElement *e))
  { cur=first;
    while (cur!=NULL)   { (*f)(cur); cur=cur->suc;}   }
 class TSortSpisok:public TSpisok // класс Сортированный список
 {protected:
   virtual int Compare(void *op1,void *op2)=0; /* абстрактная функция
                                   сравнения для процедуры сортировки */
  public:
   void Sort(void);                     // процедура сортировки
   TSortSpisok(void):TSpisok(){}  // конструктор
  ~TSortSpisok()                       // деструктор
        {puts("Уничножить сортированный список.");}
  };
 void TSortSpisok::Sort(void)
  { int swap=1;     TElement *temp;
    while (swap)
        { swap=0;
          cur=first;
          while (cur->suc!=NULL)
            { if (Compare(cur,cur->suc))
                { temp=cur->suc; cur->suc=temp->suc;   temp->suc=cur;
                  if (cur->pre!=NULL)  cur->pre->suc=temp;  else first=temp;
                  temp->pre=cur->pre;   cur->pre=temp;
                  if (cur->suc!=NULL)  cur->suc->pre=cur;   else last=cur;
                  cur=temp;
                  swap=1;      }
```

```
            else cur=cur->suc;       }
        }
    }
#include <string.h>
 class TNum: public Telement  //  класс Число
    { public:
        int num;            // числовое поле целого типа
        virtual void Print(void) { printf(" %d " ,num); }
        TNum(){}                    // конструктор по умолчанию
        TNum(int n):num(n) {}     // конструктор
        ~TNum(void) { puts(" Уничтожить число." );} // деструктор
    };
class TStr: public TNum    // класс Строка (в поле num – длина строки)
    { public:
        char st[40];            // поле символьная строка
        virtual void Print(void) { TNum::Print(); printf(" %s\n" ,st); }
        TStr(){}                // конструктор по умолчанию
        TStr(char *s):TNum(strlen(s)){
            strcpy(st,s);
            if (num>=40)st[40]='\0';
            else st[num+1]='\0'; }
        ~TStr(void) { puts(" Уничтожить строку.");}
    };
class TTabl: public TStr // класс Таблица (добавляет строку)
    { public:
        char str[20];
        virtual void Print(void) { TStr::Print();printf(" %s\n " ,str); }
        TTabl(){}  // конструктор по умолчанию
        TTabl(char *s,char *s2):TStr(s){
            strcpy(str,s2);
            if (strlen(s2)>=20)str[20]='\0';
            else str[strlen(s2)+1]='\0'; }
          ~TTabl(void) { puts(" Уничтожить таблицу.");} // деструктор
    };
  class TSSpisok:public TSortSpisok /* класс Пользовательский
                                       сортированный список*/
  { protected:
      virtual int Compare(void *op1,void *op2) /* функция сравнения для
              процедуры сортировки с явным преобразованием типа */
        { return  (((TTabl *)op1)->num)<(((TTabl *)op2)->num);}
    public:
      TSSpisok(void):TSortSpisok(){}
    ~TSSpisok(void){puts(" Уничтожить с/список." );}
  };
```

```
void Show(TElement *e) //процедура для передачи в процедуру ForEach
  { e->Print();}
TSSpisok N;   // объект класса Сортированный список
void main(void)
 { int k;   char str[40];   char str2[20];
  TElement *p; //  указатель на базовый класс Telement
   // цикл формирования списка из объектов классов TNum, TStr, TTabl
  while(printf(" Введите число:"),scanf(" %d",&k)!=EOF)
      { p=new TNum(k);
        N.Add(p);
       printf(" Введите строку:");scanf(" %s",str);
        printf(" Введите строку 2:");scanf(" %s",str2);
        p=new TTabl(str,str2);
        N.Add(p);
        printf(" Введите строку:");scanf(" %s",str);
        p=new TStr(str);
        N.Add(p);     }
    puts(" \nВведены элементы:");
    N.ForEach(Show);  // вывести элементы списка
    N.Sort();                  // сортировать
    puts(" \nПосле сортировки:");
    N.ForEach(Show);       getch();
    puts(" \nУдаление чисел");
    while ((p=N.Del())!=NULL)  { p->Print();    delete(p); }
     puts(" \nКонец."); }
```

Второй способ выполнения поэлементной обработки реализуется через определение итератора или класса итераторов, подходящего для данного вида контейнера. При помощи итераторов программист может управлять контейнером, не зная фактических типов, используемых для идентификации элементов. Несколько ключевых компонентных функций позволяют программисту найти концы последовательности элементов (раздел 1.7).

Использование шаблонов классов для проектирования контейнеров. Очень часто для реализации контейнера используются параметризованные классы или шаблоны, в том числе и стандартные, описанные в библиотеке CLASSLIB. Они также могут работать с объектами одного класса или иерархии классов. В последнем случае в качестве параметра в шаблон передается имя класса-родителя.

Пример 3.39. Контейнер на основе шаблона.

Рассмотрим программу, использующую шаблон Динамический массив для организации динамического массива, хранящего объекты классов Число и Строка (рис. 3.7).

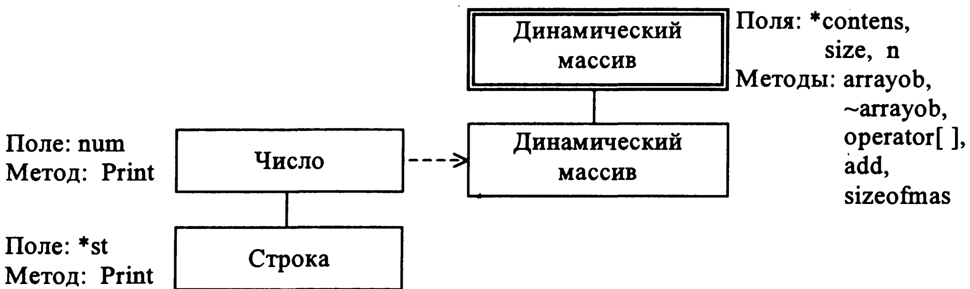


Рис. 3.7. Иерархия классов для примера 3.39

```
#include <string.h>
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
template < class type> // объявление шаблона класса с параметром type
class arrayob    // начало описания класса с именем arrayob
{ type **contents;   // массив указателей на объекты типа type
  int size;          // максимальное количество объектов в массиве
  int n;             // реальное количество объектов в массиве
public:
   arrayob(int number)  {contents=new type * [size=number];}
  ~arrayob ();
 void add(type *p)
     { if(n==size)cerr<<" выход за пределы"; // добавление элементов
       else {contents[n]=p; n++;} }
   type & operator [] (int x)    //  итератор массива объектов
    { if ((x<0)||(x>=size)) { cerr <<" ошибочный индекс "<<x<<endl;x=0;}
      return *contents[x]; }
   int sizeofmas(){return n;} // возвращает реальный размер массива
};
template < class type> arrayob< type >::~arrayob () // описание деструктора
 {for(int i=0;i<size;i++) delete contents[i];}
class TNum            // класс  Число
 { public:    int num;
       virtual void Print(void) { cout<<num<<"  "; }
       TNum(){cout<<" Введите число"<<endl;  cin>>num;}
       TNum(int n):num(n) {}
       virtual ~TNum(void) { cout<<"Уничтожить число."<<endl;}
 };
class TStr:public TNum  // класс Строка
 { public:    char *st;
```

```
        virtual void Print(void) { TNum::Print(); cout<<st<< " ";}
        TStr();                              // конструктор по умолчанию
        TStr(char *s):TNum(strlen(s)) // конструктор с параметрами
            {st=new char[num+1];strcpy(st,s); st[num]='\0';  }
        virtual ~TStr(void) { cout<<"Уничтожить строку."; delete [] st;}
  };
 TStr::TStr():TNum(40)
    {  cout<<"введите строку"<<endl;
         st=new char[num+1];   cin>>st;
         num=strlen(st);  st[num+1]='\0';    }
arrayob<TNum> ob_a(5);  /* массив из 5 указателей на объекты иерархии
                                          TNum */
void main()
{  int i;
   for(i=0;i<5;i++)    // поместить 5 объектов
        if (i/2*2==i) ob_a.add(new TNum);  // поместить Число
        else ob_a.add(new TStr);               // поместить Строку
   cout<<" содержимое контейнера "<<'\n';
   for (i=0;i<ob_a.sizeofmas();i++) ob_a[i].Print();
   getch(); }
```

Шаблон оперирует с указателями на объекты иерархии. Для вызовов методов Print и деструкторов классов иерархии используем механизм сложного полиморфизма.

Вопросы для самопроверки

1. Что такое класс в C++? Какие существуют способы ограничения доступа к компонентам класса? Как и где они используются? Чем отличается описание компонентных функций внутри и вне определения класса?

2. Сформулируйте особенности конструкторов и деструкторов классов C++? Что такое неинициилизирующий конструктор и как он отличается от конструктора без параметров? Когда использование неинициализирующего конструктора необходимо?

3. Что такое копирующий конструктор? Назовите случаи, когда использование такого конструктора необходимо.

4. Как описывается производный класс? Что такое множественное и виртуальное наследование?

5. Как определяется доступность компонент базового класса в производном классе? Какова последовательность подключения конструкторов и деструкторов базового и производного классов?

6. Назовите виды полиморфизма в C++. Определите понятие виртуальных и абстрактных функций. Что такое абстрактный класс? Назовите особенности использования абстрактного класса.

7. Что такое дружественные функции и дружественные классы? Как определить дружественные функции? Где и как они используются?

8. Что такое переопределение операций? Какие операции можно переопределять? Определите понятие функции-оператора. Чем отличаются компонентные и внешние функции-операторы?

9. Какие сложности возникают при работе с динамическими объектами? Что такое виртуальный деструктор и каковы особенности его использования?

10. Что такое шаблон? Определите понятие шаблона функции и шаблона класса. Приведите примеры применения шаблонов классов.

11. Сопоставьте понятия «параметризованные» и «контейнерные» классы? Чем определяется выбор того или иного типа классов в конкретном случае?

12. Составьте программу, обеспечивающую работу со структурой данных типа стек.

4. СОЗДАНИЕ ПРИЛОЖЕНИЙ WINDOWS

Создание программных средств для семейства операционных систем Win32 имеет целый ряд особенностей, без знакомства с которыми невозможно рассмотрение объектных моделей Delphi и C++ Builder. Прежде всего приложение Windows взаимодействует с пользователем и с операционной системой путем отправки и получения сообщений, поэтому основу любого приложения составляет цикл обработки сообщений. Сообщения, полученные приложением, диспетчируются и передаются для обработки так называемым оконным функциям, которые обеспечивают выполнение требуемых действий. Такое построение обработки в программе получило название событийного программирования. Кроме того, как правило, среды, в которых разрабатываются приложения Windows, используют визуальную технологию разработки интерфейсов.

4.1. Семейство операционных систем Windows с точки зрения программиста

Операционная система MS DOS предоставляла программисту минимальный набор средств. Она включала файловую систему, систему управления памятью, систему управления процессами и незначительный набор сервисных функций, облегчающих, например, выполнение операций ввода-вывода в простейших случаях. Вместе с этим, будучи однопрограммной и, соответственно, предоставляя программе все ресурсы системы в монопольном режиме, MS DOS позволяла программисту «напрямую», минуя операционную систему, управлять техническими средствами (рис. 4.1). Причем, при прямом управлении во многих случаях можно было получить существенно большее быстродействие при меньших затратах памяти. В результате достаточно часто программы MS DOS используют прямое управление устройствами. Например, большинство графических программ MS DOS напрямую работают с видеопамятью, причем часть из них также самостоятельно поддерживает клавиатуру и мышь.

Программирование на уровне аппаратных средств помимо достоинств имеет и существенные недостатки. Программа, непосредственно управляющая устройствами:

1) монопольно использует устройства и потому не может выполняться в мультипрограммной среде;

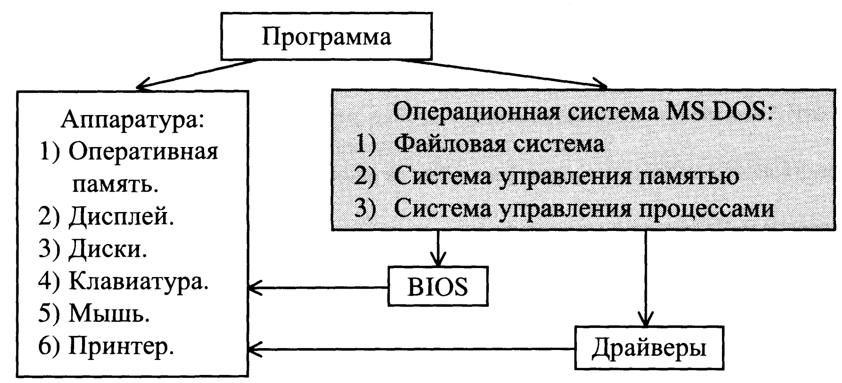


Рис. 4.1. Взаимодействие программы с MS DOS и программно-аппаратным обеспечением IBM PC

2) ограниченно переносима (или просто настроена на единственный тип устройства, например, работает только с VGA монитором);

3) требует значительного времени и трудовых ресурсов на разработку, так как программирование на столь низком уровне очень трудоемкий процесс.

Операционные системы типа Windows существенно отличаются от операционной системы MS DOS не только с точки зрения пользователя, но и с точки зрения разработчика приложений. При их разработке были максимально учтены недостатки предшествующей системы.

В настоящее время в основном распространены операционные системы Windows, получившие обобщенное название Win32: Windows'95, Windows'98, Windows NT.

От прочих Windows эти системы отличаются тем, что все они, правда, в различной степени, используют 32-битную адресацию, что является шагом вперед по сравнению с 16-битной адресацией, используемой предыдущими версиями Windows.

Между собой системы Win32 различаются следующим образом: Windows'95 и Windows'98 являются последовательными версиями операционной системы, предназначенной для широкого круга пользователей. В отличие от Windows NT, предназначенной для пользователей-профессионалов и изначально создававшейся как полноценная операционная система, Windows'95 и Windows'98 не предъявляют жестких требований ни к оборудованию, ни к программному обеспечению, но и не гарантируют той степени надежности, которую обеспечивает Windows NT.

В Windows'98 по сравнению с Windows'95 исправлены обнаруженные ошибки, несколько изменен внешний вид интерфейса, который теперь рассчитан на широкое использование InterNet, и добавлены некоторые функции.

Основными достоинствами операционных систем серии Win32 являются:

1. Системная поддержка виртуального пространства памяти до 4 Гбайт для каждого приложения Win32.

Использование 32-битных адресов позволяет адресовать до 4 Гбайт памяти. В Win32 предполагается, что каждое приложение Win32 выполняется в своем виртуальном пространстве памяти размером 4 Гбайта, в котором младшие 2 Гбайта зарезервированы за операционной системой, а старшие 2 Гбайта – отведены программе. Виртуальное пространство памяти организуется за счет использования специально выделенной области жесткого диска, которая проецируется на реальную оперативную память постранично по мере надобности. При современном быстродействии технических средств это относительно несильно тормозит выполнение операций, одновременно позволяя программисту не беспокоиться о наличии свободной оперативной памяти.

2. Возможность одновременной работы с несколькими приложениями и/или несколькими функциями приложений (многозадачность).

Реальная многозадачность, т.е. одновременное выполнение различных задач, требует наличия многопроцессорной системы, в которой каждая задача может выполняться на своем процессоре. В обычных однопроцессорных системах многозадачность реализуется за счет разделения времени процессора между задачами. Дисциплины разделения времени бывают разные. В Win32 используется вытесняющая многозадачность, подразумевающая, что управление между процессами передается по истечении некоторого заранее определенного интервала времени (кванта) по сигналу таймера, в отличие от системы Win16, где использовалась корпоративная многозадачность, при которой передача управления выполнялась по инициативе самих приложений, что позволяло приложению «захватить» процессор, остановив выполнение других приложений.

Помимо разделения времени на уровне приложений, когда каждое приложение считается отдельным процессом, системы Win32 поддерживают мультизадачность на уровне фрагментов приложений. В этом случае приложение может организовывать несколько потоков, разделение времени между которыми будет выполняться наравне с процессами приложений.

3. Стандартный графический интерфейс с пользователем.

Отсутствие единообразного удобного пользователю эргономичного интерфейса, безусловно, являлось существенным недостатком MS DOS. Стандартный и достаточно удобный интерфейс Windows, в основе которого лежит модель «рабочего стола», существенно облегчает пользователю работу с системой. При этом экран дисплея рассматривается как поверхность рабочего стола, а окна приложений – как листы бумаги на нем. Так же, как на рабочем столе, можно перекладывать «бумаги», помещая одни окна «поверх» других. Организуя «рабочее место», пользователь может изменять местоположение и размеры окон, обеспечивая себе максимальные удобства.

Программисту же наличие стандарта и соответствующих библиотек интерфейсных компонент существенно упрощают разработку программных продуктов.

4. Независимость программ от аппаратуры (в том числе – универсальная графика).

При написании программы программист больше не должен заботиться о переносимости. Практически любое приложение Win32 может выполняться на любой допустимой для указанных операционных систем конфигурации технических средств. Все вопросы, связанные с различиями подключенных устройств, решаются на уровне специальных Windows-драйверов, обеспечивающих максимальную идентичность выполнения операций с устройствами различных типов. При подключении новых устройств в систему включаются специальные программы – драйверы, обеспечивающие взаимодействие этих устройств с Windows.

5. Возможность обмена данными между приложениями.

Обмен реализуется специальным OLE (*Object Linking end Embedding* – «связывание и внедрение объектов») – механизмом, который позволяет включать документы одного приложения в другое (в результате получается документ, названный составным). *Составной документ* может содержать два типа объектов:

– связанные – основной документ содержит ссылку на документ другого приложения, соответственно, при изменении объекта его приложением изменится и составной документ;

– внедренные, в этом случае копия объекта становится частью составного документа, и никакое изменение оригинала объекта не переносится в составной документ.

Использование OLE-механизма позволяет, например, вставлять в документ Word рисунки, выполненные в Paint, или таблицы Excel.

6. Совместимость с ранее разработанным программным обеспечением.

Далеко не последним достоинством Windows является то, что она может выполнять подавляющее большинство программ MS DOS на виртуальной машине MS DOS, что позволяет не отказываться от привычных программных продуктов, разработанных до появления Windows.

Итак, согласно основной концепции, в операционной системе типа Windows программы взаимодействуют только с операционной системой, не имея возможности напрямую обращаться к аппаратуре (рис. 4.2). Управление техническими средствами осуществляется через специальный интерфейс API (*Application Program Interface* – «программный интерфейс приложения») – набор из нескольких сотен функций, выполняющих все системно-зависимые действия, такие как выделение памяти, вывод на экран и т.д. Эти функции также отвечают за разделение ресурсов системы между различными приложениями при их одновременном запуске, автоматически обеспечивая возможность работы приложения в мультипрограммной среде.

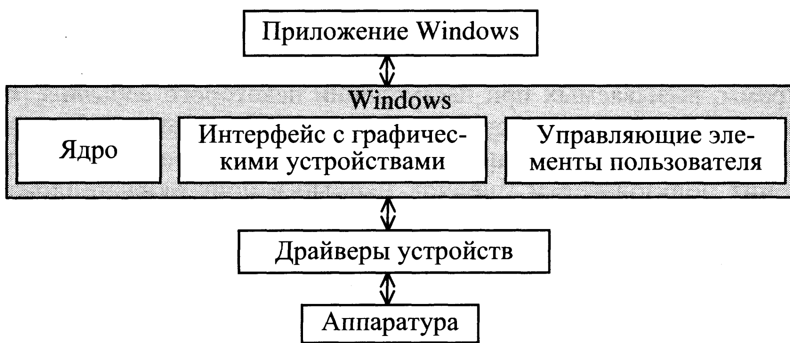


Рис. 4.2. Взаимодействие приложения Windows с операционной системой

Функции API организованы в динамически загружаемые и разделяемые всеми работающими программами библиотеки DLL (*Dynamic Link Library* – «библиотеки динамического связывания»). Особенность этих библиотек заключается в том, что коды используемых приложениями функций не включаются в исполняемый модуль. Вместо них в модуль включаются ссылки на соответствующие функции. В процессе выполнения приложение просто передает управление нужной функции уже загруженной в память библиотеки, определяя ее адрес по соответствующей таблице. Если запускаемое приложение собирается использовать функции из отсутствующей в оперативной памяти библиотеки, то эта библиотека загружается в память. Неиспользуемые библиотеки из памяти выгружаются.

Применение DLL позволяет существенно уменьшить объем приложений и увеличить эффективность использования оперативной памяти.

В основу Windows положен п р и н ц и п с о б ы т и й н о г о у п р а в л е н и я. Чтобы разобраться, что это такое, вспомним, как организован вычислительный процесс в MS DOS. При выполнении в MS DOS программа, получив управление, далее полностью контролирует вычислительный процесс: по мере необходимости запрашивает данные, выполняет расчеты и выводит результаты в соответствии с реализуемой ею последовательностью обработки (рис. 4.3). При этом пользователь большую часть времени ожидает результатов выполнения программ.

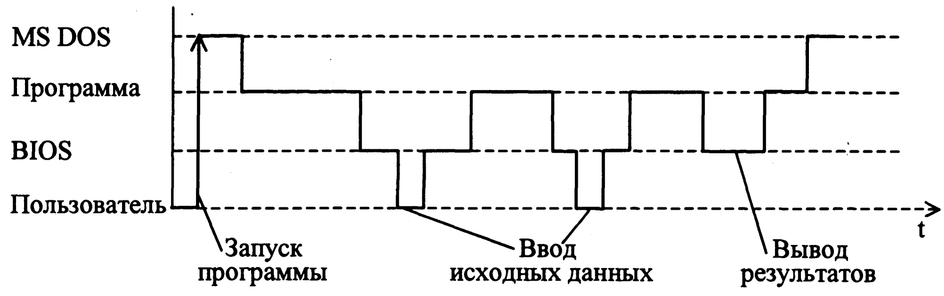


Рис. 4.3. Процесс выполнения программы в MS DOS

Принцип событийного управления (или программирования по событиям) Windows предполагает, что приложение должно состоять из нескольких подпрограмм, вызываемых при наступлении некоторого *события*, которым считается любое изменение в системе, касающееся данного приложения.

В качестве событий в системе, например, фигурируют: активизация приложения пользователем (щелчок мышью в поле окна приложения на экране), нажатие мышью кнопок приложения, выбор пунктов меню, ввод данных в активное приложение, изменение размеров окна приложения, получение данных от другого приложения, завершение заданного интервала времени и т.д.

Помимо внешних по отношению к приложению событий, перечисленных выше, могут фиксироваться и внутренние события, когда приложение должно обработать изменение собственного состояния, например, получение одним окном приложения данных от другого.

О наступлении того или иного события, на которое необходимо «отреагировать», приложение информируется посылкой соответствующих *сообщений.*

Запросы пользователя обычно вводятся в систему с использованием клавиатуры или мыши. При изменении состояния этих устройств Windows формирует сообщение и передает его активному приложению, инициируя, таким образом, выполнение некоторых действий.

В процессе работы приложение, в свою очередь, может генерировать сообщения Windows другим приложениям и самому себе, инициируя все новые и новые события. Таким образом, *все взаимодействие приложения с окружением выполняется через передачу сообщений.* Последовательность событий при таком подходе, как правило, заранее неизвестна, поэтому части приложения, которые называются *обработчиками сообщений*, взаимодействуют только через передачу данных, как и при ООП.

Приложение Windows, создав окно и запустив цикл обработки сообщений, возвращает управление операционной системе и в дальнейшем получает управление только по приходе сообщений, предназначенных данному приложению.

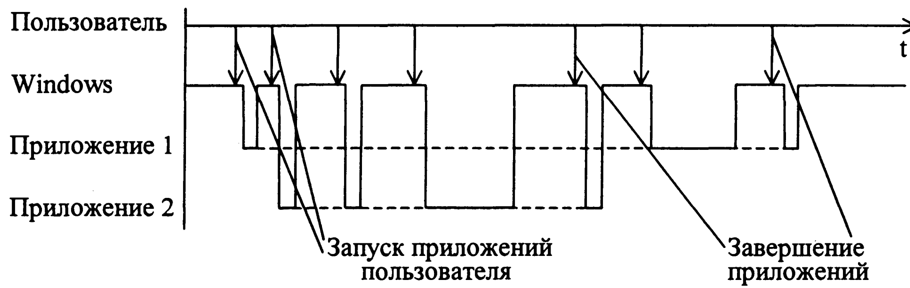


Рис. 4.4. Работа пользователя с приложениями в Windows

Более того, пользователь может вызвать одно приложение, затем второе, не завершая первого, и работать с этими приложениями попеременно (рис. 4.4) .

Для того чтобы подчеркнуть указанные выше отличия, программы, предназначенные для выполнения под управлением операционной системы типа Windows, принято называть *приложениями Windows*.

4.2. Структура приложения Windows

Каждому приложению Windows на экране соответствует окно. Окно – это прямоугольная область экрана стандартного вида (рис. 4.5), через которую пользователь взаимодействует с программой.

Хотя одно приложение может создать несколько окон, всегда имеется одно «главное» окно, при закрытии которого приложение завершает свою работу.

С точки зрения программиста, окно – это самостоятельно существующий объект, параметры которого (размеры, расположение, характеристики интерфейсных элементов и т.п.) хранятся в специальной структуре данных, а поведение определяется обработчиками сообщений, составляющими *оконную функцию*.

Таким образом, минимально любое приложение Windows состоит из двух частей: основной программы и оконной функции.

Основная программа обеспечивает функционирование приложения. При запуске она должна создать и вывести на экран главное окно приложения. При этом приложение регистрируется в системе, и для него создается очередь сообщений. Затем в основной программе запускается цикл обработки сообщений, и приложение переходит в режим ожидания сообщений. Появившиеся в очереди сообщения выбираются циклом обработки сообщений и передаются *через Windows* соответствующему окну приложения. При этом Windows вызывает нужную оконную функцию для обработки сообщения (рис. 4.6) и передает ей само сообщение в качестве параметров.

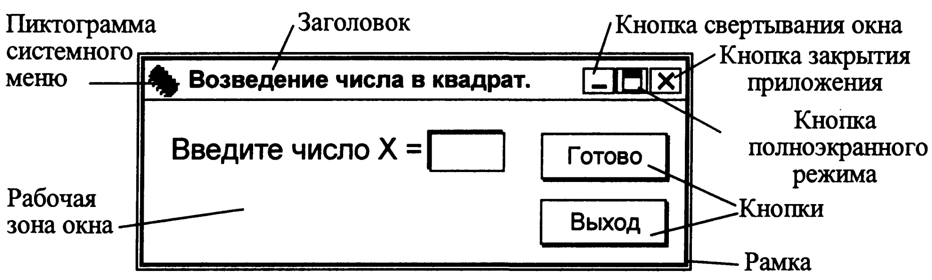


Рис. 4.5. Простейшее окно приложения Windows'95

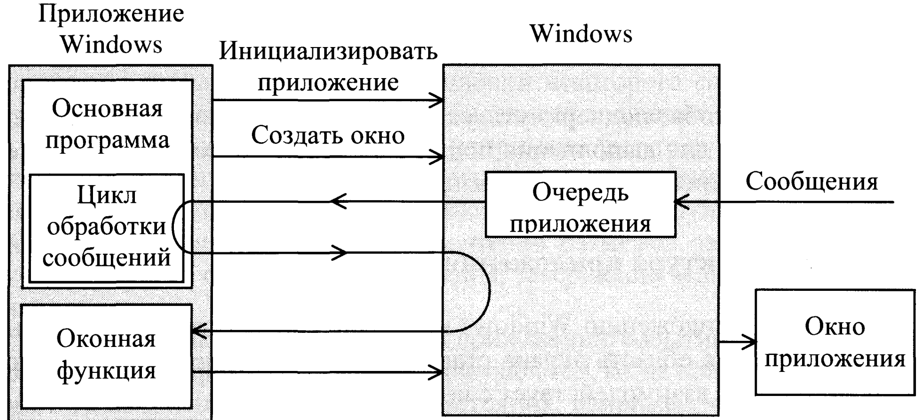


Рис. 4.6. Инициализация приложения и переход к обработке сообщений

Примечание. Такой сложный механизм обработки сообщений в первую очередь связан с тем, что программа должна регулярно возвращать управление Windows для планирования мультипрограммной обработки.

Все полезные для пользователя действия приложения программируются в виде «реакций окна» на получаемые сообщения.

Как уже говорилось выше, весь обмен информацией между объектами системы (между приложениями, между приложением и операционной системой, между основной программой и оконными функциями) осуществляется через посылку сообщений.

В Windows определено множество типов сообщений. Каждое из них кодируется уникальным 32-разрядным числом, которому ставится в соответствие стандартное имя, например, **WM_CHAR** (посылается активному приложению при нажатии и отпускании клавиш на клавиатуре), **WM_PAINT** (посылается окну при необходимости его перерисовки), **WM_CLOSE** (посылается окну при необходимости его закрытия), **WM_TIMER** (посылается приложению при истечении указанного интервала времени) и т. д.

Помимо уникального номера сообщение включает два 32-разрядных параметра, содержащих дополнительную информацию, характер которой зависит от типа сообщения.

Например, при изменении состояния клавиатуры (нажатии или отпускании клавиши) драйвер клавиатуры инициирует посылку сообщения **WM_CHAR** *активному* приложению (приложению, окну которого принадлежит фокус ввода). При этом через параметры передаются ASCII-код символа, количество повторений, определяемое при удержании клавиши в нажатом положении, scan-код. В качестве дополнительной информации сообщается, была ли клавиша нажата или отпущена, была ли клавиша нажата до формирования сообщения, нажата ли одновременно клавиша Alt и т.д.

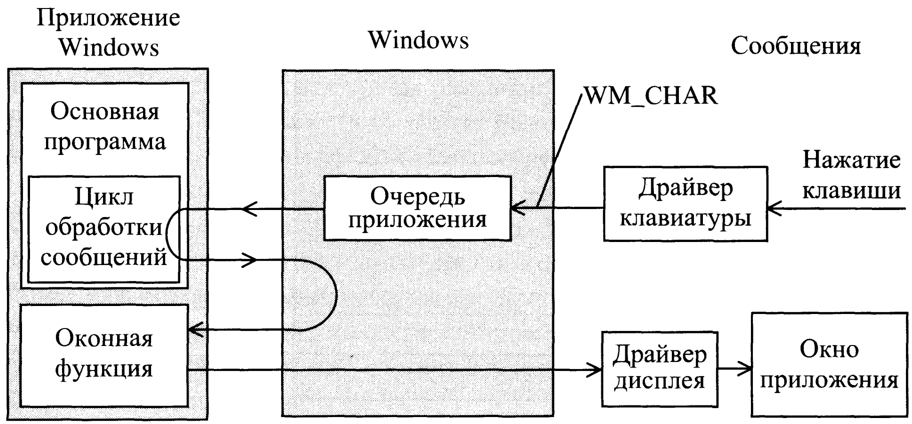


Рис. 4.7. Обработка ввода символа и его отображение в окне

Приложение, получив сообщение, сопровождающееся столь подробной информацией, может выбрать те данные, которые ему необходимы и, например, вывести этот символ в окно приложения (рис. 4.7).

Заметим, что для выполнения операций, связанных с управлением программно-аппаратными ресурсами (в том числе и для организации вывода информации в «свое» окно), обработчики сообщений должны обращаться к операционной системе.

В отличие от сообщений от клавиатуры, сообщения от мыши передаются тому приложению, на поле окна которого находится курсор мыши, т.е. могут быть отправлены как активному, так и неактивному приложению. Получив сообщение от мыши, неактивное приложение становится активным, т.е. получает фокус ввода, и теперь вся вводимая с клавиатуры информация будет передаваться этому приложению.

Однако не все сообщения попадают в функцию окна приложения через очередь сообщений. Приложение Windows одновременно манипулирует сообщениями трех типов:

внешними (от Windows и других приложений) и *внутренними* (от самого приложения) *сообщениями*, поступающими через очередь сообщений;

прямыми вызовами оконных функций (такой вызов также оперирует информацией, форматированной под сообщение);

прямыми вызовами методов не оконных объектов приложения (в отличие от предыдущих случаев передаваемое сообщение как в ранних ООП системах не форматировано под сообщение).

Так, сообщения поддержки окна, формируемые Windows, передаются оконной функции напрямую.

Примером может служить сообщение **WM_DESTROY** (рис. 4.8), которое формируется при закрытии пользователем окна приложения.

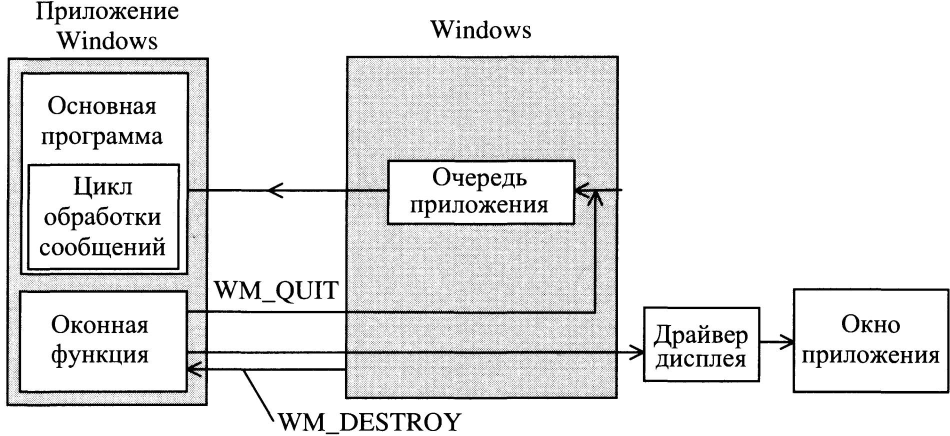


Рис. 4.8. Обработка сообщения WM_DESTROY

Это сообщение сразу передается оконной функции, которая вызывает функцию API, выполняющую закрытие окна приложения, и формирует сообщение **WM_QUIT**, помещая его в очередь приложения. При выборке этого сообщения в цикле обработки сообщений выполняется завершение цикла, и управление передается операторам завершения приложения.

4.3. Технология разработки приложений Windows в средах Delphi и C++Builder

Идея событийного управления является самостоятельной и непосредственно не связана с ООП, т. е. существуют программы, работающие по событийному принципу и написанные с использованием API вне рамок объектного подхода. Однако совмещение обеих технологий позволяет существенно упростить процесс программирования.

В настоящее время существуют несколько сред программирования «под Windows», использующих объектно-ориентированный подход и включающих мощные библиотеки объектов. В качестве примеров можно назвать среды разработки Visual C++, Delphi, C++Builder, которые позволяют сравнительно легко создавать сложнейшие приложения Windows. Эти пакеты автоматизируют многие операции создания приложения, предлагая разработчику большое количество различных шаблонов и заготовок, начиная от заготовки будущего приложения.

Примечание. Visual C++ является наиболее универсальным из названных трех пакетов, но и, соответственно, самым сложным. Он использует развитую объектную модель C++ и имеет более мощную по сравнению с Delphi и C++Builder библиотеку объектов. Интерфейс и средства Visual C++ ориентированы на профессиональные разработки высокого уровня и далее в настоящем курсе рассматриваться не будут.

Delphi и C++Builder используют идентичные среды и одну и ту же библиотеку объектов VCL (*Visual Component Library* – библиотека визуальных компонент). Практически эти среды различаются только языком разработки: Delphi использует язык на основе Object Pascal, а C++Builder, как указано в названии, C++. Используемые объектные модели также во многом похожи.

Любое приложение, созданное в этих средах, состоит как минимум из трех объектов: объекта-приложения, объекта-формы (окна) и объекта-экрана. В большинстве случаев разработчик использует те варианты объекта-приложения и объекта-экрана, которые ему предоставляются средой. Основное внимание в процессе разработки сосредоточивается на создании объектов-форм и соответствующих оконных функций. При этом широко используются стандартные классы библиотеки VCL.

Все окна приложения строятся на базе класса формы TForm. В терминах библиотеки VCL *форма* – это окно, которое может содержать другие визуальные (т.е. имеющие графический образ на экране – кнопки, метки, строчный редактор и т.д.) и невизуальные компоненты. Соответствующие компонентам объекты в качестве объектных полей включаются в класс, порождаемый от TForm (см. рис. 4.15).

При входе в среду разработчику предлагается заготовка будущего приложения, которая состоит из заготовки формы и заготовки оконной функции. Заготовка оконной функции высвечивается в окне текста программы (рис. 4.9).

Имеется и заготовка проекта, предусматривающая создание и инициализацию объекта приложения (класс TAplication), ответственного за создание главного окна приложения и цикл обработки сообщений. Заготовка формы (объект класса TForm1, наследуемого от TForm) уже «умеет» выполнять некоторые стандартные действия окна приложения, т.е. содержит методы обработки некоторых сообщений, например, закрытие приложения при нажатии на кнопку «Завершить приложение».

Заготовка приложения уже содержит все, что необходимо для создания минимального приложения, и может быть запущена на выполнение, правда при отсутствии полезного эффекта.

Предлагаемые разработчику стандартные классы визуальных компонент, так же как и TForm, уже включают методы – обработчики сообщений, наиболее типичных для этих компонент.

Для включения специфической обработки приложения во всех обработчиках предусмотрены вызовы методов, которые могут быть добавлены при разработке конкретного приложения. Эти методы получили название *обработчиков событий*, так как их вызов осуществляется при фиксации в процессе обработки сообщения некоторого *события*.

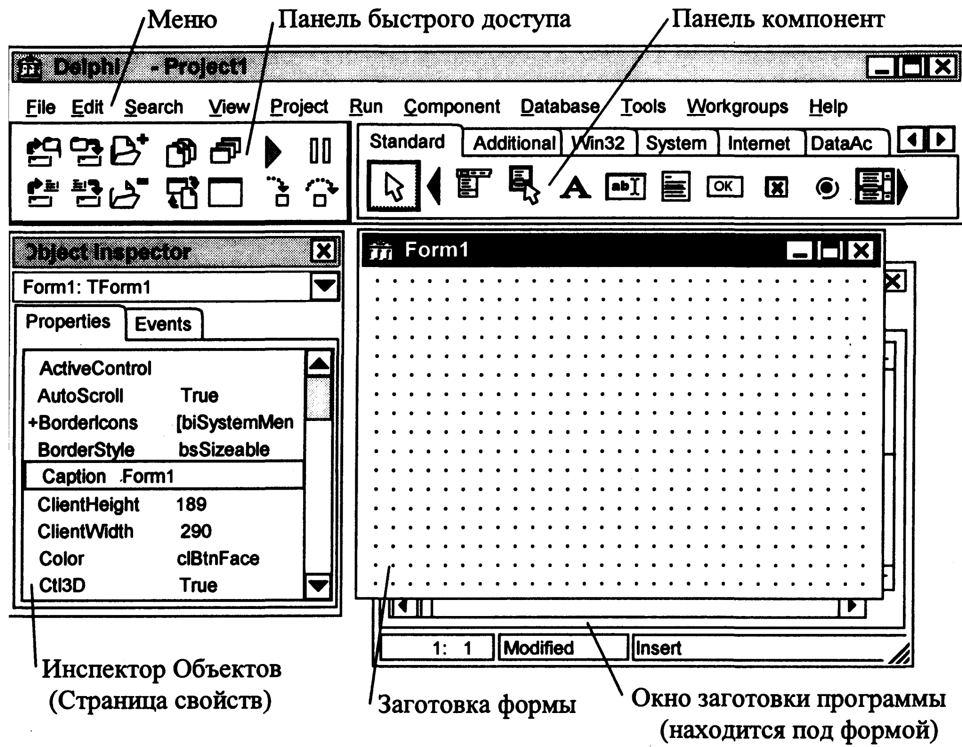


Рис. 4.9. Вид экрана при входе в Delphi

Так класс TForm включает метод обработки сообщения **WM_CHAR**. В ависимости от того, какая именно кнопка мыши была нажата, этот метод может ызывать:

1) обработчик события **OnKeyPress** – при нажатии любой клавиши, оторой соответствует код ASCII (в основном – алфавитно-цифровая часть лавиатуры);

2) обработчики событий **OnKeyDown** и **OnKeyUp** – соответственно, при ажатии и отпускании любой клавиши, включая и те, которым соответствует од ASCII (рис. 4.10).

Таким образом, в Delphi и C++Builder обработка *сообщений* заменена бработкой *событий*, при этом программист в большинстве случаев збавляется от необходимости расшифровки сообщений системы.

Перечень событий, предусмотренных для визуальных компонент, можно осмотреть в Инспекторе Объектов, поместив нужный компонент из меню омпонент на форму, выделив его щелчком мыши и переключившись в нспекторе Объектов на страницу Events (события). Так, например, для класса Form предусмотрена возможность подключения обработчиков следующих обытий:

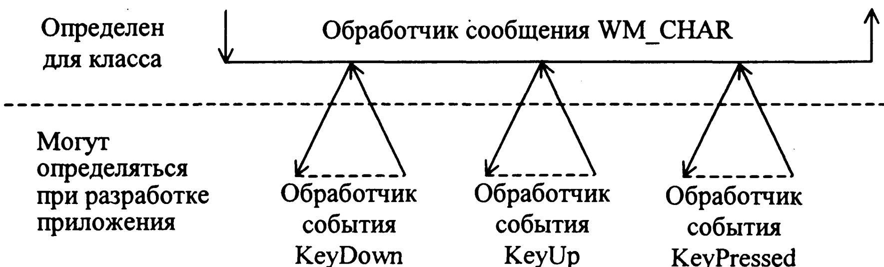


Рис. 4.10. Подключение обработчиков событий при обработке сообщения

при изменении состояния формы:

OnCreate – в начальной стадии создания формы – используется при необходимости задания параметров (цвет или размер);

OnActivate – при получении формой фокуса ввода (окно становится активным и ему адресуется весь ввод с клавиатуры);

OnShow – когда форма (окно) становится видимой;

OnPaint – при необходимости нарисовать или перерисовать форму;

OnResize – при изменении размеров формы на экране;

OnDeactivate – при потере формой фокуса ввода (окно становится неактивным);

OnHide – при удалении формы с экрана (окно становится невидимым);

OnCloseQuery – при попытке закрыть форму – обычно используется для создания запроса-подтверждения необходимости закрытия окна;

OnClose – при закрытии формы;

OnDestroy – при уничтожении формы;

от клавиатуры и мыши:

OnKeyPressed – при нажатии клавиш, которым соответствует код ASCII;

OnKeyDoun, OnKeyUp – при нажатии и отпускании любых клавиш, включая те, которым соответствует код ASCII;

OnClick, OnDblClick – при обычном и двойном нажатии клавиш мыши соответственно;

OnMouseMove – при перемещении мыши (многократно);

OnMouseDown, OnMouseUp – при нажатии и отпускании клавиш мыши;

при перетаскивании объекта мышью:

OnDragDrop – в момент опускания объекта на форму;

OnDragOver – в процессе перетаскивания объекта над формой (многократно);

другие;

OnHelp – при вызове подсказки;

OnChange – при изменении содержимого компонент.

Список событий более простых компонент существенно короче, например, метка (класс TLabel) может реагировать на 9 событий:

OnClick, **OnDblClick**, **OnMouseDown**, **OnMouseUp** – события мыши;

OnDragDrop, **OnDragOver**, **OnEndDrag**, **OnStartDrag** – события при перетаскивании объекта мышью.

В зависимости от типа события, для каждого обработчика предусмотрен соответствующий список параметров. При необходимости программист может добавить свой обработчик каких-либо сообщений, возможно, предусмотрев в нем подключение обработчиков своих событий.

Конструирование окна приложения выполняется с использованием визуальной технологии, т.е. проектировщик, создавая проект окна, переносит на него мышью нужные элементы с панели компонент, имея возможность сразу же визуально оценить полученный вариант. Параллельно, используя Инспектор Объектов, он может задать или изменить некоторые параметры и/или определить обработчики событий для включаемых компонент.

Таким образом, в Delphi и C++ Builder разработчику предоставляются средства визуального проектирования интерфейса и некоторые вспомогательные средства (в виде библиотеки классов) разработки содержательной части задания.

Процесс создания приложений Windows в указанных средах включает те же этапы, что процесс разработки объектных программ для MS DOS, но выполнение этих этапов имеет некоторую специфику.

Прежде всего, процесс создания программы имеет ярко выраженный итерационный характер. Обычно вначале выполняется анализ, проектирование и реализация интерфейса или даже его части. А затем программа поэтапно наращивается до получения окончательного варианта.

Существуют и особенности выполнения самих этапов разработки. Рассмотрим два примера.

Пример 4.1. Приложение «Возведение чисел в квадрат» (вариант 1). Пусть необходимо разработать приложение, которое должно осуществлять последовательный ввод вещественных чисел (с проверкой правильности вводимых чисел) и возведение вводимых чисел в квадрат.

А н а л и з.

1. Разработку начинаем с анализа задачи с целью определения основных состояний интерфейса.

Для данной задачи можно использовать интерфейс с тремя состояниями:

а) исходное состояние, в котором приложение ожидает ввода числа, которое необходимо возвести в квадрат;

б) состояние вывода результата;

в) возможное состояние вывода сообщения об ошибочном вводе исходного числа.

2. Для каждого состояния продумываем внешнее представление (экранную форму), т.е. вид окна приложения в каждом случае (рис. 4.11).

Состояние ввода

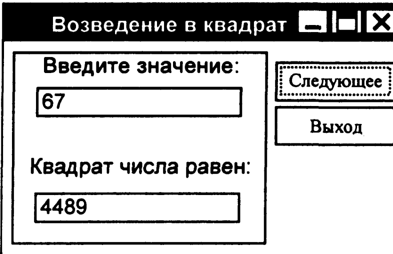
Возведение в квадрат

Введите значение:

Следующее

Выход

Состояние вывода результата



Состояние вывода сообщения об ошибке

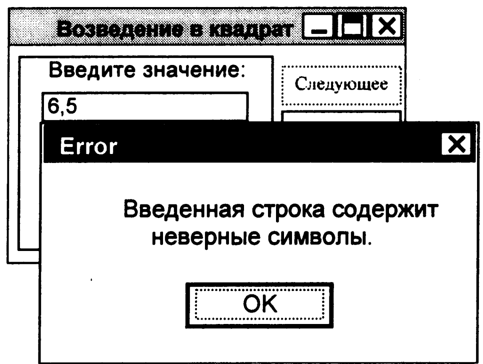


Рис. 4.11. Экранные формы

3. Разрабатываем граф состояний интерфейса, отражающий возможные варианты смены состояний (рис. 4.12).

4. Выполняем объектную декомпозицию интерфейсной и предметных частей приложения. Интерфейсная часть включает два объекта: главное окно приложения и окно сообщения об ошибке. Предметная часть очень проста и может быть реализована без выделения объектов (рис. 4.13).

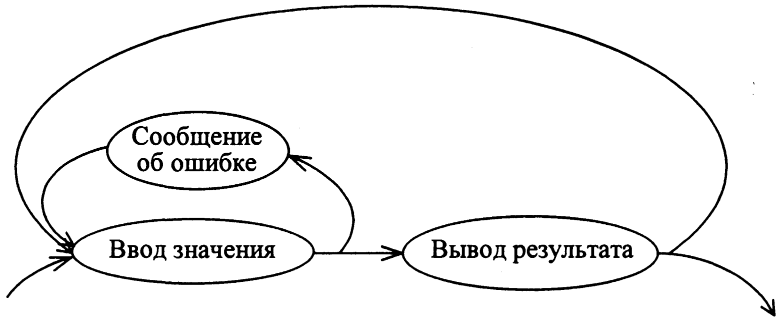


Рис. 4.12. Граф состояний интерфейса

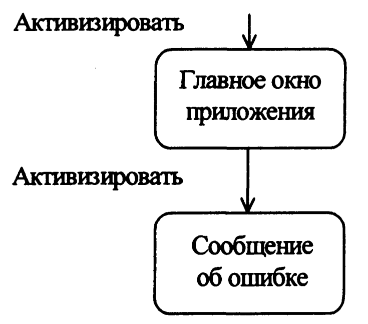


Рис. 4.13. Объектная декомпозиция

Проектирование.

1. Логическое проектирование или разработка структуры классов выполняется с использованием среды разработки Delphi.

а) Вначале организуем новый проект и, используя визуальную технологию, создаем окно главной формы (рис. 4.14).

При этом среда автоматически строит описание класса TForm1, наследуемого от библиотечного класса TForm, и включает в него объектные поля визуальных компонент, руководствуясь эскизом экранной формы.

б) С помощью Инспектора Объектов настраиваем параметры формы и компонент следующим образом:

Form1:
Name:=MainForm;
Caption:='Возведение числа в квадрат';
Label1:
Name:='InputLabel';
Caption:='Введите значение';
Label2:
Name:=OutPutLabel;
Caption:='Квадрат значения равен:';
Edit1:
Name:=InputEdit;
Edit2:
Name:=OutPutEdit;
Enable:=false;
ReadOnly:=true;

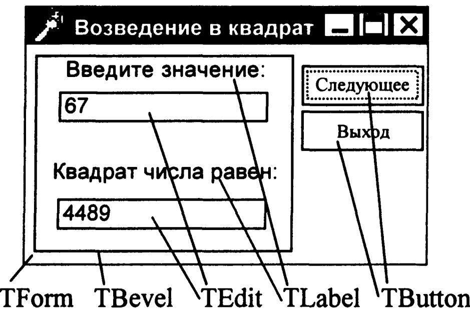


Рис. 4.14. Проектирование главного окна приложения

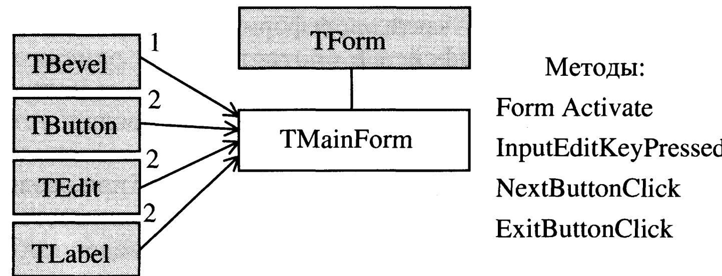


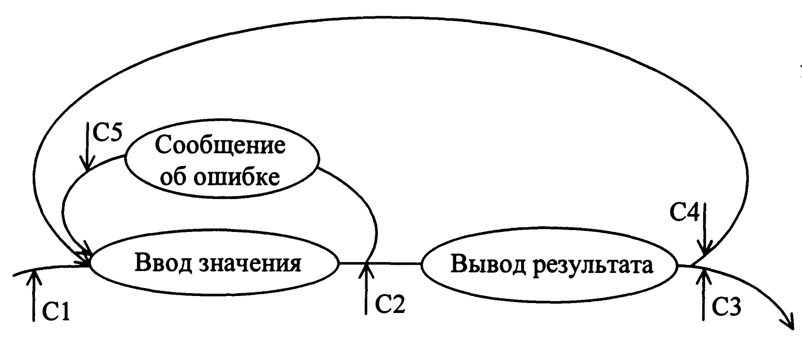
Рис. 4.15. Диаграмма классов интерфейса приложения

Button1:
Name:=NextButton;
Caption:='Следующее';
Button2:
Name:=ExitButton;
Caption:='Выход';

Диаграмма класса TMainForm, построенного при визуальным проектировании, представлена на рис. 4.15.

в) Исходя из возможных вариантов (списка событий задействованных компонент), определяем события, по которым будет происходить смена состояний интерфейса (рис. 4.16).

Параллельно определяем, какие основные операции должны быть выполнены при смене состояний интерфейса.



События:

C1 – активация формы
C2 – нажатие клавиши Enter
при вводе значения
C3 – нажатие кнопки Выход
C4 – нажатие кнопки Следующее
C5 – нажатие кнопки ОК

Рис. 4.16. Определение множества обрабатываемых событий

Обработчик события С1 – «активация формы» (**OnFormActivate**) должен выполнять настройку интерфейса в соответствии с экранной формой, изображенной на рис. 4.11, *а* (прятать метку результата и поле его вывода, запрещать нажатие кнопки Следующее, чистить поле ввода и устанавливать на него курсор).

Обработчик события С2 – «нажатие клавиши Enter при вводе значения» (**OnKeyPressed** для поля ввода) должен проверять правильность ввода. При правильном вводе далее он должен вычислять квадрат введенного значения и выводить результат в экранную форму, изображенную на рис. 4.11, *б* (соответственно, запрещая ввод значений, разрешая нажатие кнопки Следующее и показывая метку результата и результат). При неправильном вводе он должен выводить сообщение об ошибке, используя экранную форму, изображенную на рис. 4.11, *в*.

Обработчик события С3 – «нажатие кнопки Выход» (**On Click** для кнопки Выход) должен закрывать окно приложения.

Обработчик события С4 – «нажатие кнопки Следующее» (**On Click** для кнопки Следующее) должен вновь настраивать интерфейс в соответствии с экранной формой, изображенной на рис. 4.11, *а*.

На основании этой информации на следующем этапе разрабатываются алгоритмы соответствующих процедур – обработчиков указанных событий.

г) Решаем, что окно сообщения об ошибке отдельно проектировать не будем: для его отображения используем специальную функцию MessageDlg.

2. **Физическое проектирование** или разбивку программного продукта на модули среда Delphi выполнит автоматически, поместив класс формы в отдельном модуле.

Эволюция.

Используя шаблоны, предоставляемые средой, создаем обработчики событий, выбранных при разработке интерфейса:

```
Unit MainUnit;
Interface
  Uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
          logs, StdCtrls, ExtCtrls;
  Type TMainForm = class(TForm)        {Класс «Главная форма»}
      Bevel1: TBevel;                  {рамка}
      NextButton, ExitButton: TButton; {кнопки Следующее и Выход}
      InputLabel, OutPutLabel: TLabel; {метки ввода и вывода}
      InPutEdit, OutPutEdit: TEdit;    { редакторы ввода и вывода }
      procedure FormActivate(Sender: TObject); {обработчик события
          «активация формы»}
      procedure InPutEditKeyPress(Sender: TObject; var Key: Char);
          {обработчик события «нажатие клавиши Enter при вводе
          значения»}
```

```
        procedure NextButtonClick(Sender: TObject); {обработчик события
              «нажатие кнопки Следующее»}
        procedure ExitButtonClick(Sender: TObject); {обработчик события
              «нажатие кнопки Выход»}
      end;
    Var  MainForm: TMainForm;  {объявить объект класса TMainForm}
Implementation
  {$R *.DFM}
  {обработчик события «активация формы»}
 Procedure TMainForm.FormActivate(Sender: TObject);
  Begin
    NextButton.Enabled:=false;   {дезактивировать кнопку Следующее}
    InPutEdit.ReadOnly:=false;   {разрешить ввод в Редактор ввода}
    InPutEdit.Clear;             {очистить Редактор ввода}
    InPutEdit.Enabled:=true;     {активизировать Редактор ввода}
    InPutEdit.SetFocus;          {установить фокус ввода}
    OutPutLabel.Visible:=false;  {скрыть Редактор вывода}
    OutPutEdit.Visible:=false;   {скрыть Метку редактора вывода}
  End;
  {обработчик события «нажатие клавиши Enter при вводе значения»}
  Procedure TMainForm.InPutEditKeyPress(Sender: TObject;
                                               var Key: Char);
   Var x:real;Code:integer;
   Begin
    if Key=#13 then  { если нажата клавиша Enter, то}
      begin
      Key:=#0;  {предотвратить звуковой сигнал при вводе символа перехода
                 на следующую строку в однострочном редакторе}
        Val(InPutEdit.Text,x,Code); {преобразовать введенную строку в
                                   число}
        if Code=0 then     {если преобразование прошло успешно, то}
          begin
            InputEdit.ReadOnly:=true;  {запретить ввод в Редактор ввода}
            InputEdit.Enabled:=false;  {дезактивировать Редактор ввода}
            OutPutLabel.Visible:=true; {показать Метку вывода}
            OutPutEdit.Visible:=true;  {показать Редактор вывода}
            OutPutEdit.Text:=floattostr(sqr(x)); {вывести результат в
                                                 Редактор вывода}
            NextButton.Enabled:=true; {активировать кнопку Следующее}
            NextButton.SetFocus;      {установить фокус на кнопку
                                                 Следующее}
          end
        else       {иначе, если введено не число, то}
```

```
            MessageDlg(' Значение содержит недопустимые символы.',
              mtError, [mbOk], 0); {показать окно Сообщение об ошибке}
      end
   End;
   {обработчик события «нажатие кнопки Следующее»}
   Procedure TMainForm.NextButtonClick(Sender: TObject);
    Begin
       FormActivate(NextButton); {вернуться к исходным настройкам
                                          интерфейса}
    End;
   {обработчик события «нажатие кнопки Выход»}
   Procedure TMainForm.ExitButtonClick(Sender: TObject);
     Begin Close;     {закрыть окно приложения и завершить его}
     End;
End.
```

Основная программа (проект) в процессе разработки с использованием среды Delphi создается автоматически и выглядит следующим образом:

```
Program Example1;
   Uses  Forms,   MainUnit in 'MainUnit.pas' {MainForm};
        {$R *.RES}
     Begin
          Application.Initialize;            {инициализация приложения}
          Application.CreateForm(TMainForm, MainForm); {создание
                                                   формы}
          Application.Run;           {запуск цикла обработки сообщений}
     End.
```

В процессе компиляции программы будут участвовать следующие файлы:

*.dpr – файл проекта программы – исходный текст на Pascal (создается автоматически);

*.pas – файлы, содержащие исходные модули программы (в том числе модули форм);

*.dfm – текстовые файлы описания форм.

В результате – будет получен файл .exe программы и целый ряд вспомогательных файлов.

Пример 4.2. Приложение «Возведение чисел в квадрат» (вариант 2). Если использовать для разработки той же программы среду C++ Builder, то первые два этапа будут выполняться идентично, хотя описание класса TMainForm будет автоматически создано на C++ в файле MainUnit.h:

```
#ifndef MainUnitH
#define MainUnitH
```

```
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
class TMainForm : public TForm
{
__published:        // компоненты, управляемые средой (объекты VCL)
  TLabel *InputLabel;
      TLabel *OutPutLabel;
      TBevel *Bevel1;
      TEdit *InputEdit;
      TEdit *OutPutEdit;
      TButton *NextButton;
      TButton *ExitButton;
      void __fastcall FormActivate(TObject *Sender);
      void __fastcall InputEditKeyPress(TObject *Sender, char &Key);
      void __fastcall NextButtonClick(TObject *Sender);
      void __fastcall ExitButtonClick(TObject *Sender);
private:            // объявления пользователя
public:             // объявления пользователя
      __fastcall TMainForm(TComponent* Owner);
};
extern PACKAGE TMainForm *MainForm;
#endif
```

Затем на третьем этапе будут программироваться обработчики событий, в результате чего будет получен файл MainUnit.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include <math.h>
#include <stdlib.h>
#include «MianUnit.h»
#pragma package(smart_init)
#pragma resource ' *.dfm'
TMainForm *MainForm;
__fastcall TMainForm::TMainForm(TComponent* Owner)
     : TForm(Owner)  { }    // конструктор формы
// обработчик события «активация формы»
void __fastcall TMainForm::FormActivate(TObject *Sender)
{
 NextButton->Enabled=false; // дезактивировать кнопку Следующее
 InputEdit->ReadOnly=false; // разрешить ввод в Редактор ввода
```

```
  InputEdit->Enabled=true;       // активизировать Редактор ввода
  InputEdit->Clear();            // очистить Редактор ввода
  InputEdit->SetFocus();         // установить фокус на Редактор ввода
  OutPutEdit->Visible=false;     // скрыть Редактор вывода
  OutPutLabel->Visible=false;    // скрыть Метку редактора вывода
}
// обработчик события «нажатие клавиши Enter при вводе значения»
void __fastcall TMainForm::InputEditKeyPress(TObject *Sender,
                      char &Key)
{ float x;
  if (Key=='\r')      // если введен символ «конец строки», то
   {Key='\0';         // предотвратить выдачу звукового сигнала
     try            // выполнить, контролируя возникновение исключений
      {x=StrToFloat(InputEdit->Text);  // преобразовать в число
       InputEdit->ReadOnly=true;       // запретить ввод в Редактор ввода
       InputEdit->Enabled=false;       // дезактивировать Редактор ввода
       OutPutLabel->Visible=true;      // показать Метку вывода
       OutPutEdit->Visible=true;       // показать Редактор вывода
       OutPutEdit->Text=FloatToStr(x*x); // вывести результат в Редактор
                                           вывода
       NextButton->Enabled=true;       // активизировать кнопку Следующее
       NextButton->SetFocus();   // установить фокус на кнопку Следующее
      }
     catch (EConvertError&) //  перехватить исключение преобразования
      {TMsgDlgButtons S1; // объявить объект класса «множество»
       S1<<mbOK;           // занесения кода кнопки OK во множество S1
       MessageDlg("Строка содержит недопустимые  символы.",
            mtError, S1, 0);      // вывести окно Сообщение об ошибке
      }
   }
}
//  обработчик события «нажата кнопка Следующее»
void __fastcall TMainForm::NextButtonClick(TObject *Sender)
 { FormActivate(NextButton); //  вернуться к исходным настройкам
 }
//      обработчик события «нажата кнопка Выход»
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
  { Close();}                  // завершить работу приложения
```

Файл проекта Example1.cpp C++Builder также создает автоматически:

```
#include <vcl.h>
#pragma hdrstop
USERES("Example1.res"); // включить файл ресурсов
```

```
USEFORM(" MainUnit.cpp" , MainForm);  // включить файл формы
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
                              /* основная программа «под Windows»*/
{   try         // выполнить, контролируя возникновение исключений
     {   Application->Initialize(); // инициализация объектов VCL
         Application->CreateForm(__classid(TMainForm), &MainForm);
                              // создание главного окна приложения
         Application->Run();  // запуск цикла обработки сообщений
     }
    catch (Exception &exception)       // при возникновении исключений
     {   Application->ShowException(&exception);  // вывести сообщение
    }
    return 0;
}
```

Таким образом, программа будет компилироваться из файлов следующих типов:

.cpp – файлы, содержащие исходные модулей программы (в том числе обработчики событий форм) и файл проекта программы – исходный текст на C++ (обычно создается автоматически);

.bpr – текстовый файл, содержащий инструкции по компиляции и сборке проекта (make-файл, обычно создается автоматически);

.dfm – текстовые файлы описания форм (генерируются средой);

.h – заголовочные файлы, содержащие объявления классов;

.hpp – заголовочные файлы, содержащие описание компонентов VCL.

В результате компиляции и компоновки системы также как в Delphi получается файл .exe и вспомогательные файлы:

.res – двоичные файлы ресурсов;

.tds – символьная таблица отладчика.

Вопросы для самоконтроля

1. Перечислите основные различия между операционными системами MS DOS и группой Win32. Сформулируйте, как каждое из них влияет на разрабатываемое программное обеспечение.

2. Что собой представляет приложение Win32? Назовите основные компоненты приложения.

3. Определите понятия: «сообщение», «цикл обработки сообщений», «оконная функция», «обработчик сообщений». Поясните, каким образом происходит выполнение приложения в операционных системах Win32.

4. Что такое визуальная среда программирования? Что связывает между собой среды Delphi и C++Builder? Что такое «событие»? Как связаны между собой сообщения и события?

5. Назовите последовательность разработки программных систем в указанных средах.

5. ОБЪЕКТНАЯ МОДЕЛЬ DELPHI PASCAL

Объектная модель Delphi Pascal по сравнению с моделью, использованной Borland Pascal 7.0., является более полной. Помимо уже описанных в главе 2 средств ООП она включает:

– ограничение доступа к полям и методам за счет определения собственного интерфейса к каждому полю класса (*пять типов секций при объявлении класса, свойства*);

– более развитые механизмы реализации полиморфных методов (*абстрактные, динамические методы*);

– средства работы с метаклассами (*переменные метаклассов, методы классов, механизм RTTI*);

– возможность делегирования методов (*указатели на методы*) и т. д.

5.1. Определение класса

Все классы, используемые в Delphi (уже определенные в библиотеках классов и создаваемые разработчиком для конкретного приложения), наследуются от класса *TObject.*

Формат описания нового класса выглядит следующим образом:

Type <имя объявляемого класса> = **class**(<имя родителя>)

private	<скрытые элементы класса>
protected	<защищенные элементы класса>
public	<общедоступные элементы класса>
published	<опубликованные элементы класса>
automated	<элементы, реализующие OLE-механизм>

end;

Имя родителя указывать не обязательно, по умолчанию считается, что, если имя родителя не указано, то класс непосредственно наследуется от TObject.

Внутри описания класса выделяется до пяти секций.

Секция *private* содержит *внутренние* элементы, обращение к которым возможны только в пределах модуля (а не класса, как в C++), содержащего объявление класса.

Секция *protected* содержит *защищенные* элементы, которые доступны в пределах модуля, содержащего объявление класса, и внутри потомков класса.

Секция *public* содержит *общедоступные* элементы, к которым возможно обращение из любой части программы.

Секция *published* содержит *опубликованные* элементы, которые по ограничению доступа аналогичны public. Для визуальных компонент, вынесенных на панель компонент, информация об элементах, размещенных в этой секции, становится доступной через Инспектор Объектов.

Секция *automated* содержит элементы, доступ к которым также выполняется аналогично public. Но для элементов, описанных в этой секции, генерируется дополнительная информация, используемая OLE. Секцию имеет смысл объявлять для потомков класса TAutoObject.

Потомки класса могут менять область доступности всех элементов родительского класса, кроме элементов, объявленных в секции private, так как последние им не доступны.

Основной особенностью объектов Delphi является то, что они всегда являются *динамическими*, т.е. размещаемыми в динамической области памяти. Соответственно, переменная типа класса по смыслу представляет собой указатель на объект, однако, по правилам Delphi, при работе с этой переменной операция разыменования «^» не используется. Вызовы же конструктора и деструктора в такой модели становятся обязательными, так как конструктор выполняет размещение объекта в памяти, а деструктор – выгрузку из нее.

Рассмотрим пример, демонстрирующий различие старой и новой объектных моделей Pascal.

Borland Pascal 7.0:

```
Type pTNum = ^TNum;
     TNum = Object
       n: integer;
       constructor Init (an:integer);
      end;
Constructor TNum.Init;
 begin
    n:=an;
 end;        . . .
  Var p:pTNum;
  Begin
    New(p, Init(5));
    WriteLn(p^.n);
    Dispose(p) . . . . .
```

Delphi:

```
Type
  TNum = class
    public  n:integer;
   constructor Create (an:integer);
 end;
Constructor TNum.Create;
   begin      inherited Create;
    n:=an;
 end;         . . .
  Var A:TNum;
              . . .
    A:=TNum.Create(5);
    WriteLn(A.n);
    A.Destroy. . . . .
```

Чтобы подчеркнуть изменение функций конструктора и деструктора в объектной модели Delphi, для них предлагается использовать другие имена: *Create* (создать) – для конструктора и *Destroy* (уничтожить) – для деструктора. Если конструктор или деструктор для создаваемого класса не указывается, то

они наследуются от родительского класса или через него от более далеких предков. В конечном счете, классу всегда должны быть доступны конструктор и деструктор класса TObject, которые отвечают за размещение и выгрузку объектов.

Конструктор Create класса TObject вызывает специальный метод InstanceSize для определения размера памяти, необходимой для размещения объекта, запрашивает область памяти требуемого размера, используя процедуру NewInstance, и *инициализирует поля нулевыми значениями*, используя процедуру InitInstanse. Адрес области памяти конструктор-функция возвращает в качестве результата.

При создании собственного конструктора для разрабатываемого класса необходимо перед программированием специфических операций, предусматриваемых для конструирования объектов данного класса, обращаться к наследуемому (inherited) конструктору:

```
Constructor TNum.Create;
  begin
    inherited Create;   {наследуемый конструктор}
    n:=an;
  end;
```

Правильно построенная цепочка выполнения наследуемых конструкторов в иерархии должна восходить к конструктору TObject, который и обеспечит размещение объектов данного класса в памяти.

Примечание. Если конструктор вызывается для уже существующего объекта, то выполняются только явно описанные действия, память не выделяется и не обнуляется.

Деструктор Destroy класса TObject вызывает метод CleanUpInstance для корректного завершения работы с длинными строками и другими сложными структурами данных. Затем он обращается к методу InstanceSize для определения размера объекта и освобождает память. В классе TObject деструктор объявлен виртуальным, так как с одной стороны в том же классе существует метод Free (см. ниже), который вызывает деструктор, с другой стороны сам деструктор в производных классах может быть переопределен. Следовательно, при переопределении деструктора в классе его необходимо объявлять как метод, перекрывающий виртуальный (раздел 5.2):

```
Destructor Destroy; override;
```

В теле самого деструктора необходимо вызвать родительский деструктор, обеспечивая формирование цепочки деструкторов, восходящей к деструктору

класса TObject, который обеспечит освобождение памяти, занимаемой объектом. Обычно это делается после программирования всех специфических операций по уничтожению данного объекта, например:

```
Destructor TNum.Destroy;
begin
   . . .     {выполнение специфических операций над объектом}
     inherited Destroy;        {вызов родительского деструктора}
end;
```

При работе с объектами в Delphi необходимо постоянно помнить о том, что все объекты Delphi размещены в динамической памяти, а соответствующие переменные на самом деле являются указателями на объекты.

Так, например, операция присваивания объектов соответствует операции *копирования адреса*, а не самого объекта. В табл. 5.1 поясняется, что происходит при выполнении некоторых операций.

Таблица 5.1. **Особенности работы с объектами в Delphi**

Операция	Результат	Пояснение
Var A, B:TNum;	A [nil] B [nil]	Определение переменных типа TNum. Память отводится только для размещения указателей
A:=TNum.Create;	A [ ]→Объект, B [nil]	Конструирование объекта. Объект размещается в памяти, адрес заносится в указатель
B:=A;	A [ ]→Объект, B [ ]→Объект	Адрес из указателя A переписывается в B. Теперь оба указателя содержат адрес одного объекта
B.Destroy;	A [ ]→?, B [ ]→?	Уничтожается объект по адресу B. Память, отведенная под него, освобождается. Однако указатели A и B по-прежнему содержат этот адрес
A.Destroy; или **B.Destroy;** или **A .n**	A [ ]→?, B [ ]→?	Теперь любая попытка обращения по указателям A или B приведет к ошибке

На первом шаге в соответствии с описанием создаются переменные А и В. При создании они обнуляются, т.е. в них записывается значение **nil**. На втором шаге выполнено конструирование объекта А. Теперь указатель А содержит адрес этого объекта. На третьем шаге выполняется операция присваивания над объектами, в результате которой указатель В теперь также содержит адрес того же объекта. На четвертом шаге уничтожается объект В. Теперь и А и В указывают на уже освобожденное поле памяти, которое может использоваться системой для других целей. Попытка повторного освобождения того же участка памяти с использованием любого указателя приведет к возникновению ошибки адресации. Попытка обращения к полю или методу объекта с использованием этих указателей также приведет к возникновению ошибки адресации.

Чтобы уменьшить количество ошибок адресации, можно использовать вместо метода Destroy метод Free. Этот метод прежде, чем освобождать память, проверяет, был ли создан объект в памяти (сравнивает адрес, записанный в переменной, со значением **nil**) и, если объект не создавался, то память не освобождается. Однако во многих случаях, например, для ситуации, представленной в табл. 5.1, замена Destroy на Free не поможет избежать ошибки адресации: в данном варианте объект был создан, и адрес в переменной отличен от **nil**.

Проблема корректного уничтожения объектов в сложных программах может оказаться далеко не тривиальной. В простых же программах, когда вопрос об экономии памяти не стоит, обычно объекты создают в начале программы, а уничтожают – при ее завершении, часто используя с этой целью секции инициализации и завершения модуля.

Пример 5.1. Определение класса (графический редактор «Окружности» – вариант 1). Пусть требуется разработать приложение, позволяющее рисовать на специальном поле окна приложения окружности. Местоположение центра окружности должно определяться щелчком левой клавиши мыши. Радиус окружности и цвет контура должны настраиваться.

Разрабатываемое приложение легко декомпозируется на два объекта: окно приложения и изображаемая в нем окружность (или круг, так как в процессе рисования внутреннее поле окружности заполняется точками цвета фона). Результат декомпозиции приведен на рис. 5.1.

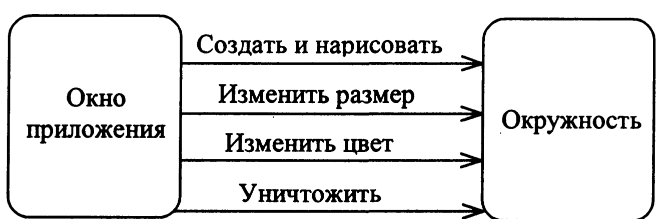


Рис. 5.1. Объектная декомпозиция приложения «Окружности»

Окно должно содержать интерфейсные элементы, позволяющие выполнять настройку параметров круга и саму процедуру рисования. Оно проектируется на базе класса TForm с использованием стандартных визуальных компонент. Предлагаемый внешний вид окна с указанием типов используемых визуальных компонент представлен на рис. 5.2.

Компоненты, использованные для формирования внешнего вида окна приложения, настраиваются с помощью Инспектора Объектов следующим образом:

Form1:
Name:=MainForm;
Caption:='Графический редактор
"Окружность"';
Bevel1:
Name:= Bevel;
Image1:
Name:=Image;
Label1:
Name:=rLabel;
Caption:='Радиус круга';

Edit1:
Name:=rEdit;
Text:='';
UpDown1:
Name:=UpDown;
Associate:=rEdit;
Button1:
Name:=ColorButton;
Button2:
Name:=ExitButton;

Рис. 5.2. Вид главного окна приложения «Окружности»

Кроме указанных компонент приложение будет использовать стандартный диалог выбора цвета. При визуальном проектировании формы этот диалог в виде условного изображения добавляется в форму и в нужный момент программно инициируется. Имя этого компонента также настраивается через Инспектор Объектов (Name:=ColorDialog).

Интерфейс может находиться в двух состояниях: ожидание действий пользователя, связанных с рисованием, и ожидание выбора цвета (в диалоге выбора цвета). Граф состояний интерфейса, содержащий привязку к событиям, доступным при разработке приложения в Delphi, приведен на рис. 5.3.

Обработчик события С1 (**MainFormActivate**) должен устанавливать цвет фона и исходный цвет рисования.

Обработчик события С2 (**ImageMouseDown**) определив, что нажата именно левая клавиша, должен рисовать круг с заданным радиусом текущим цветом контура и центром в точке местонахождения курсора мыши.

Обработчик события С3 (**ColorButtonClick**) должен активизировать диалог выбора цвета. После завершения этого диалога он должен проверять, существует ли объект класса Окружность, и если такой объект существует, то перерисовывать его установленным цветом.

Событие С4 будет обрабатываться диалогом выбора цвета.

Обработчик события С5 (**UpDownClick**) должен проверять, существует ли объект класса Окружность, и если такой объект существует, то стирать старое изображение и выводить новое с изменением размера.

Обработчик события С6 (**ExitButtonClick**) должен завершать приложение.

Диаграмма класса TMainForm, наследуемого от стандартного класса TForm и включающего объектные поля стандартных классов Delphi, приведена на рис. 5.4, *а*.

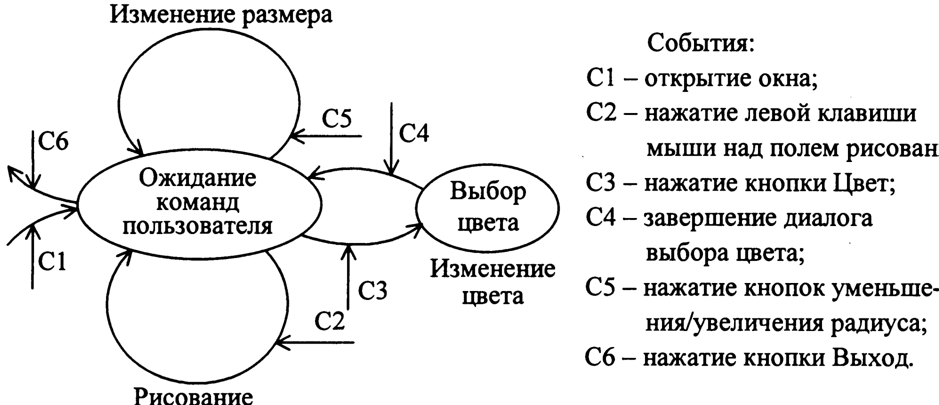


Рис. 5.3. Граф состояний интерфейса

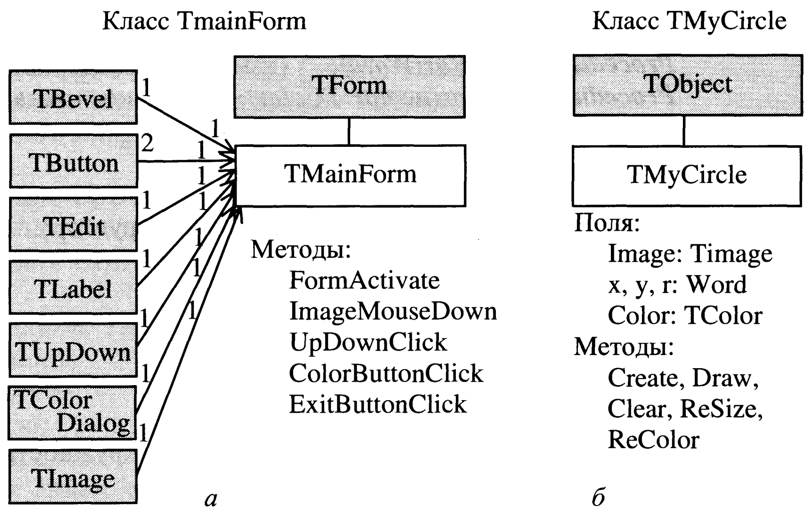


Рис. 5.4. Диаграммы разрабатываемых классов

Класс TMyCircle (рис. 5.4, *б*) в соответствии с правилами Delphi наследуется от TObject. Он добавляет поля для хранения координат центра окружности (x,y), ее радиуса r, цвета Color и имени компонента Image, на холсте которого нарисована окружность. Набор методов в основном определяется перечнем сообщений, на которые должен реагировать объект данного класса (см. рис. 5.2). Дополнительно переопределим метод Create, чтобы использовать его для инициализации полей объекта при создании. Кроме этого, целесообразно определить метод Clear для стирания нарисованной окружности при изменении ее размера или цвета.

Определим тип доступа для всех компонентов класса: все поля и метод Clear, к которому не требуется доступ извне, объявим в секции private (внутренние), а все остальные компоненты – в секции public.

Объявление класса TMyCircle выполним в отдельном модуле Circul:

```
Unit Circul;
 Interface
   Uses  extctrls,Graphics;
   Type TMyCircle=class
             private
             x, y, r:Word;    {координаты центра и радиус окружности}
             Color:TColor;  {цвет}
              Image:TImage; {поле для рисования}
             Procedure Clear; {стирание окружности}
           public
               Constructor Create(aImage:TImage; ax,ay,ar:Word;
                               aColor:TColor); {конструктор}
```

```
                Procedure Draw;                {рисование}
                Procedure ReSize(ar:Word);     {изменение размеров}
                Procedure ReColor(acolor:TColor);{изменение цвета}
        end;
 Implementation
  Constructor TMyCircle.Create;
   Begin  inherited Create;  {вызвать наследуемый конструктор}
          Image:=aImage;     {инициализировать поле}
          x:=ax;  y:=ay;  r:=ar;   Color:=aColor;
   End;
 Procedure TMyCircle.Draw;
   Begin
     Image.Canvas.Pen.Color:=Color;  {задать цвет пера}
     Image.Canvas.Ellipse(x-r, y-r, x+r, y+r); {нарисовать окружность}
   End;
 Procedure TMyCircle.Clear;
 Var TempColor:TColor;
  Begin
     TempColor:=Color;        {сохранить цвет пера}
     Color:=Image.Canvas.Brush.Color;{фиксировать цвет фона}
     Draw;                    {нарисовать цветом фона – стереть }
     Color:=TempColor;        {востановить цвет пера}
  End;
 Procedure TMyCircle.ReSize;
   Begin  Clear;  r:=ar;  Draw;  End;
 Procedure TMyCircul.ReColor(aColor:TColor);
   Begin  Clear;  Color:=aColor;  Draw;  End;
End.
```

Теперь переходим к программированию обработчиков событий, при наступлении которых должны выполняться основные операции приложения. Ниже приводится текст модуля Main, соответствующего форме TMainForm.

```
Unit Main;
 Interface
 Uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
         Dialogs, ExtCtrls, StdCtrls, ComCtrls;
 Type
  TMainForm = class(TForm)
   Bevel: TBevel;            {рамка}
   Image: TImage;            {поле рисования}
   ColorButton, ExitButton: TButton;  {кнопки}
   rEdit: TEdit;             {редактор ввода радиуса}
   rLabel: TLabel;           {метка ввода радиуса}
```

```
    UpDown: TUpDown; {компонент «инкремент/декремент»}
    ColorDialog: TColorDialog; {диалог выбора цвета}
    Procedure FormActivate(Sender: TObject);
    Procedure ImageMouseDown(Sender: TObject; Button: TMouseButton;
                             Shift: TShiftState; X, Y: Integer);
    Procedure UpDownClick(Sender: TObject; Button: TUDBtnType);
    Procedure ColorButtonClick(Sender: TObject);
    Procedure ExitButtonClick(Sender: TObject);
  end;
 Var  MainForm: TMainForm;
Implementation
  {$R *.DFM}
 Uses Circul;
 Var C:TMyCircle;
 Procedure TMainForm.FormActivate(Sender: TObject);
   Begin Image1.Canvas.Brush.Color:=clWhite;  {установить белый фон}
       Image1.Canvas.Pen.Color:=clBlack;  {установить цвет рисования}
   End;
 Procedure TMainForm.ImageMouseDown(Sender: TObject;
              Button: TMouseButton;  Shift: TShiftState; X, Y: Integer);
   Begin
     if Button=mbLeft then        {если нажата левая клавиша мыши}
      begin
        C.Free;                   {если объект создан, то уничтожить его}
        C:=TMyCircle.Create(Image1,X,Y,strtoint(rEdit.Text),
                    Image1.Canvas.Pen.Color);  {конструировать}
        C.Draw;          {изобразить объект с заданными параметрами}
      end;
   End;
 Procedure TMainForm.UpDown1Click(Sender: TObject;
                                     Button: TUDBtnType);
   Begin
     if C<>nil then                          {если объект создан, то}
        C.ReSize(strtoint( rEdit.Text));  {перерисовать с другим радиусом}
   End;
 Procedure TMainForm.ColorButtonClick(Sender: TObject);
   Begin
     if ColorDialog.Execute then  {если выполнен диалог выбора цвета, то}
        Image1.Canvas.Pen.Color:=ColorDialog.Color;  {установить цвет}
     if C<>nil then                            {если объект создан, то}
     C.ReColor(Image1.Canvas.Pen.Color); {перерисовать другим цветом}
   End;
 Procedure TMainForm.ExitButtonClick(Sender: TObject);
```

```
    Begin  Close;  End;
Initialization
Finalization     C.Free;        {если объект создан, то уничтожить его}
End.
```

Из приведенного текста видно, что конкретный объект класса TMyCircle создается при нажатии левой клавиши мыши. Причем, если объект уже существовал, то он уничтожается. Соответствующая уничтоженному объекту окружность на холсте сохраняется, но теряется связь изображения и объекта. После этого мы больше не можем менять цвет и размер этой окружности. Таким образом, одна и та же переменная используется многократно для хранения адресов различных объектов при рисовании. Последний используемый объект уничтожается при закрытии формы, когда выполняются операции, записанные в секции finalization.

5.2. Особенности реализации переопределения методов

В Delphi Pascal реализованы более сложные механизмы переопределения методов. Прежде всего в нем различают: виртуальные методы, динамические методы и абстрактные методы. Кроме того, в последних версиях языка реализована параметрическая перегрузка как обычных процедур и функций, так и методов. Использование соответствующих средств позволяет создавать существенно более мощные классы.

Виртуальные методы. Виртуальные методы объектной модели, используемой Delphi, практически ничем не отличаются от тех, которые определялись в Borland Pascal 7.0. Для них также реализуется механизм позднего связывания, обеспечивая возможность построения виртуальных объектов. Не изменилась и сущность реализации механизма позднего связывания через ТВМ. Изменения коснулись лишь описания виртуальных методов. Теперь только самый первый виртуальный метод в иерархии описывается *virtual*, все же методы, перекрывающие этот метод, описываются со спецификатором *override*. Если же по ошибке какой-нибудь из этих методов описать virtual, то будет создано новое семейство виртуальных полиморфных методов, родоначальником которого является метод, описанный virtual по ошибке.

Примечание. Вызов конструктора выполняется следующим образом. Вначале он вызывается как метод класса и организует размещение в памяти нового объекта, а затем вызывается уже как обычный метод. Поэтому конструктор может объявляться виртуальным. Виртуальный конструктор позволяет создавать объекты различных типов, устанавливаемых во время выполнения программы.

Динамические методы. *Динамическими* в Delphi называются полиморфные виртуальные методы, доступ к которым выполняется не через

ТВМ, а через специальную таблицу динамических методов (ТДМ). Такие методы описываются, соответственно, *dymamic* – при объявлении и *override* – при переопределении.

В отличие от ТВМ, которая хранит адреса всех виртуальных методов данного класса, ТДМ хранит только адреса виртуальных методов, определенных в данном классе (рис. 5.5). Если осуществляется вызов динамического метода, определенного в предках данного класса, то его адрес отыскивается в организованных в списковую структуру описаниях классов иерархии (ТВМ – раздел 5.6). Естественно, поиск такого метода займет больше времени по сравнению с виртуальными методами. Однако ТДМ занимает меньше места, так как не хранит всех предшествующих ссылок.

Из вышесказанного следует, что использовать динамическую реализацию полиморфных виртуальных методов имеет смысл лишь в тех случаях, когда класс имеет сотни потомков, а описываемый метод вызывается крайне редко, или если метод переопределяется в каждом потомке. Во всех остальных случаях лучше использовать обычную реализацию (virtual).

Абстрактные методы. *Абстрактные* методы используются при объявлении методов, реализация которых откладывается. Такие методы в классе описываются служебным словом *abstract* и обязательно переопределяются в потомках класса.

Класс, в состав которого входят методы с отложенной реализацией, называется *абстрактным*. Создавать (конструировать) объекты абстрактных классов запрещается.

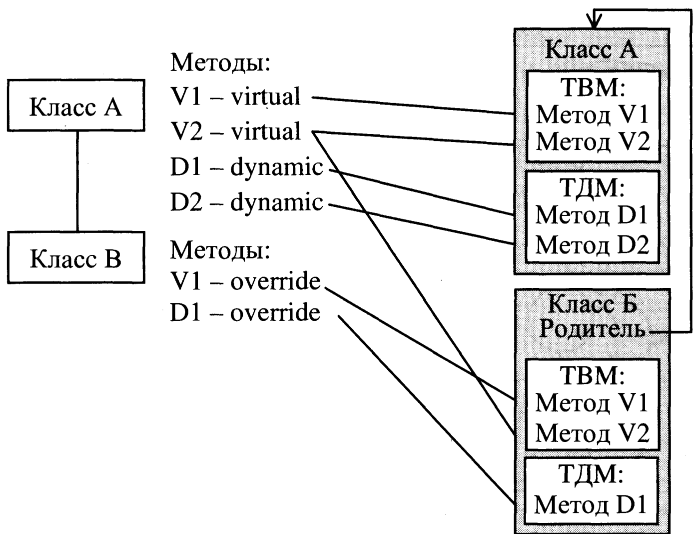


Рис. 5.5. Виртуальные и динамические методы

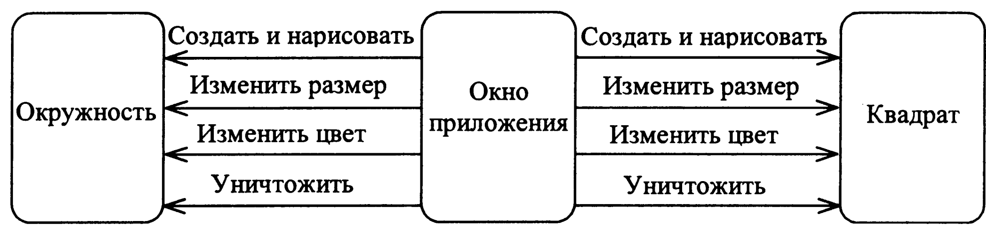


Рис. 5.6. Объектная декомпозиция приложения «Окружности и квадраты»

Пример 5.2. Использование абстрактных методов (графический редактор «Окружности и квадраты» – вариант 1). Попробуем изменить пример предыдущего раздела так, чтобы можно было рисовать не только окружности, но и, например, квадраты.

Диаграмма объектов такого приложения изображена на рис. 5.6.

Естественно, теперь интерфейс должен обеспечивать возможность выбора фигуры. С этой целью добавим в форму компонент TRadioGroup группу «радиокнопок» (рис. 5.7).

Соответственно при обработке события С2 (см. рис.5.3) необходимо учитывать тип выбранной фигуры.

Класс TMainForm будет иметь ту же структуру, что и в предыдущем примере, но в него будет добавлено поле RadioGroup типа TRadioButton.

Класс TMySquare можно построить несколькими способами.

Рис. 5.7. Вид главного окна приложения «Окружности и квадраты»

1. Класс TMySquare можно наследовать от TObject, так же как был получен классTMyCircle. В этом случае нам придется повторить программирование всех методов данного класса, что явно нецелесообразно.

2. Класс TMySquare можно наследовать от TMyCircle (рис.5.8, *а*). Тогда мы можем наследовать от TMyCircle методы Create, Clear, ReColor, ReSize, определяющие общие элементы поведения. Метод Draw необходимо объявить виртуальным, так как он вызывается из наследуемых методов и переопределен в классе-потомке. Метод Draw класса TMySquare должен быть объявлен переопределенным – override. Данный вариант наследования, принципиально применимый в конкретном случае, является не универсальным (и, в общем, не логичным: как можно «наследовать» от круга квадрат?).

3. С учетом существования двух типов объектов со сходным поведением можно создать абстрактный класс TFigure, инкапсулирующий требуемый тип поведения фигур (рис. 5.8, *б*). В этом классе нужно объявить метод Draw виртуальным (virtual) и абстрактным (abstract) и определить методы Create, Clear, ReColor, ReSize через метод Draw. Теперь мы можем наследовать от этого абстрактного класса классы, рисующие любые фигуры. Эти классы должны будут переопределять абстрактный метод Draw класса TMyFigure.

Представленный ниже текст модуля Figure включает описание иерархии классов в соответствии с рис. 5.8, *б*.

```
Unit Figure;
 Interface
  Uses  extctrls,Graphics;
  Type TMyFigure=class
     private      x,y,r:Word;      Color:TColor;      Image:TImage;
                       procedure Clear;
     public
       Constructor Create(aImage:TImage;ax,ay,ar:Word;aColor:TColor);
         Procedure Draw; virtual; abstract; {абстрактная процедура}
```


Рис. 5.8. Два варианта иерархии классов

```
    Procedure ReSize(ar:Word);
    Procedure ReColor(acolor:TColor);
    end;
   TMyCircle=class(TMyFigure)        {класс Окружность}
      public    Procedure Draw; override; {рисование окружности}
    end;
    TMySquare=class(TMyFigure)        {класс Квадрат}
      public    Procedure Draw; override; {рисование квадрата}
    end;
Implementation
  Constructor TMyFigure.Create;
    Begin
      inherited Create;
      Image:=aImage;  x:=ax;  y:=ay;  r:=ar;  Color:=aColor;
    End;
Procedure TMyFigure.Clear;
 Var TempColor:TColor;
 Begin   TempColor:=Color;
         Color:=Image.Canvas.Brush.Color;
         Draw; {нарисовать фигуру цветом фона – стереть}
         Color:=TempColor;
 End;
Procedure TMyFigure.Resize;
     Begin  Clear;  r:=ar;  Draw;  End;
Procedure TMyFigure.Recolor;
     Begin  Clear;  Color:=aColor;  Draw; End;
Procedure TMyCircle.Draw;
   Begin   Image.Canvas.Pen.Color:=Color;
           Image.Canvas.Ellipse(x-r,y-r,x+r,y+r);
   End;
Procedure TMySquare.Draw;
   Begin   Image.Canvas.Pen.Color:=Color;
           Image.Canvas.Rectangle(x-r,y-r,x+r,y+r);
   End;
End.
```

Пример демонстрирует, что использование абстрактных классов позволяет разрабатывать более универсальные и логически обоснованные иерархии.

Примечание. При создании аналогичных иерархий средствами Borland Pascal 7.0 приходилось использовать «пустые» методы, включающие только операторные скобки **begin end**. При этом компилятор не мог контролировать создание объектов абстрактных классов, предоставляя разработчику самостоятельно отыскивать ошибки данного вида.

Перегрузка методов. Начиная с версии 4, в Delphi Pascal появилась возможность перегрузки процедур и функций:

```
function Divide(X, Y: Real): Real; overload; {вариант 1}
  begin     Result := X/Y;   end;
function Divide(X, Y: Integer): Integer; overload; {вариант 2}
  begin     Result := X div Y;  end;
```

Какой конкретно вариант перегруженных процедур и функций вызван, определяется по типам фактических параметров:

```
k:=Divide(2.5, 7.3); {вызывается вариант 1}
k:=Divide(6,3);      {вызывается вариант 2}
```

Перегружать можно не только простые процедуры и функции, но и методы. *Перегруженный метод* не перекрывается как переопределенный, а *остается доступным, как любой другой метод базового класса.*

Для описания перегруженных методов используется служебное слово overload. Нужный аспект метода при его перегрузке также определяется по совпадению типов фактических параметров:

```
type
  T1 = class(TObject)
    public
      procedure Test(I: Integer);  overload;
  end;
  T2 = class(T1)
    public
      procedure Test(S: string);  overload;
  end;
  ...
Var SomeObject:T2;
  ...
  SomeObject := T2.Create;
  SomeObject.Test('Hello!');  {параметр строка – вызвается метод Test класса
                               T2}
  SomeObject.Test(7);          {параметр целое число – вызывается метод Test
                               класса T1}
```

В отличие от перегруженных методов C++ перегруженные методы Delphi Pascal не могут объявляться внутри одного класса:

```
Type
  TSomeClass = class
```

```
  public
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  { ошибка!}
  ...
```

Если перегружается виртуальный метод, то компилятор выдает предупреждение о перекрытии доступа к этому методу. Для того чтобы сообщить компилятору, что некоторый метод действительно должен перекрыть витруальный метод, используется служебное слово reintroduce.

Рассмотрим различные варианты переопределения виртуальных методов:

```
Type
 T1 = class(TObject)
  public  procedure Act1; virtual;
          procedure Act2; virtual;
          procedure Act3; virtual;
          procedure Act4; virtual;
          procedure Act5; virtual;
 end;
 T2 = class(T1)
  public
     procedure Act1;override; {определение нового аспекта виртуального
                     полиморфного метода}
     procedure Act2;virtual;  {определяется новый полиморфный метод –
                     компилятор выдает предупреждение о возможной ошибке, так
                     как при этом доступ к ранее описанному полиморфному методу
                     перекрывается}
     procedure Act3;reintroduce;virtual; {определяется новый полиморфный
                     метод – компилятор  предупреждения не выдает, так как
                     пресечение доступа документировано}
     procedure Act4;          {переопределение виртуального метода простым –
                     компилятор выдает предупреждение о возможной ошибке, так
                     как при этом доступ к виртуальному методу перекрывается}
     procedure Act5; reintroduce; {переопределение виртуального метода
                     простым – компилятор предупреждения не выдает, так как
                     пресечение доступа документировано}
 end;
var  SomeObject1: T1;
begin
  SomeObject1 := T2.Create;  {указатель на базовый класс содержит адрес
                             объекта производного класса}
  SomeObject1.Act1;  {позднее связывание – вызывается метод класса T2}
  SomeObject1.Act2;  {раннее связывание – вызывается метод класса T1}
```

```
  SomeObject1.Act3;  {раннее связывание – вызывается метод класса T1}
  SomeObject1.Act4;  {раннее связывание – вызывается метод класса T1}
  SomeObject1.Act5;  {раннее связывание – вызывается метод класса T1}
end;
```

Соответственно, описание классов, использующих перегрузку виртуальных методов, должно выполняться по следующему образцу:

```
type

T1 = class(TObject)
   public
     procedure Test(I: Integer); overload; virtual;
 end;
 T2 = class(T1)
  public
    procedure Test(S: string); reintroduce; overload;
 end;
```

При перегрузке как обычных процедур и функций, так и методов, необходимо очень осторожно обращаться с параметрами, принимаемыми по умолчанию (которые также введены начиная с версии 4 Delphi):

```
procedure Confused(I: Integer);  overload;
Begin... End;
procedure Confused (I: Integer; J: Integer = 0); overload;
Begin... End;
 ...
Confused(X); {ошибка компиляции, не определен вариант перегруженной
              процедуры}
```

Аналогичная ситуация при перегрузке методов не приводит к появлению сообщения об ошибке, просто один из перегруженных методов становится недоступным (в примере ниже недоступен перегруженный метод базового класса):

```
Type
 T1 = class(TObject)
   public
     procedure Test(I: Integer);  overload; virtual;
 end;
 T2 = class(T1)
  public
    procedureTest (I:Integer;S:string='aaa'); reintroduce; overload;
 end;
 ...
```

```
SomeObject := T2.Create;
SomeObject.Test(5,'Hello!');  {параметры целое число и строка – вызвается
                                   метод Test класса T2}
SomeObject.Test(7);        {параметр целое число – вызывается метод Test
                              класса T2 с параметром по умолчанию}
```

Перегрузить можно любой метод, в том числе и конструктор:

```
Type T1=class
       public     I:integer;
       Constructor Create;overload;
     end;
     T2=class(T1)
      public
       Constructor Create(aI:Integer);overload;
     end;
Constructor T1.Create;
Begin
  inherited Create;
  I:=10;
End;
Constructor T2.Create(aI:Integer);
Begin
  inherited Create;
  I:=aI;
End;
...
Var SomeObject:T2;
...
SomeObject:=T2.Create;  {вызывается конструктор базового класса – в поле I
                            записано 10}
SomeObject:=T2.Create(5); {вызывается конструктор производного класса – в
                              поле I записано 5}
```

5.3. Свойства

Свойство – это средство Pascal Delphi, позволяющее определять интерфейс доступа к полям класса.

В Delphi различают:

простые или скалярные свойства;

свойства-массивы;

индексируемые свойства или свойства со спецификацией index;

процедурные свойства.

Простые свойства. Описание простых свойств выполняется следующим образом (квадратные скобки в данном случае указывают, что соответствующая спецификация может быть опущена):

property <имя свойства>:<тип>
  [read <метод чтения или имя поля>**]**
  [write <метод записи или имя поля>**]**
  [stored <метод или булевское значение>**]**
  [default <константа>**];**

Спецификации, начинающиеся служебными словами read и write, определяют соответственно методы, которые должны быть использованы для чтения и записи свойства. Если метод чтения неопределен, то свойство недоступно для чтения. Если метод записи неопределен, то свойство недоступно для записи.

Вместо метода чтения или записи возможно указание имени поля. Это означает, что данному свойству соответствует поле в описании класса, куда осуществляется непосредственная запись или откуда выполняется непосредственное чтение (без использования методов).

Остальные спецификации, названные спецификациями сохранения, используются только для свойств, определяемых в секции published, т.е. используемых для «опубликованных» компонент. Булевское значение после stored (вместо которого может быть указан метод, возвращающий булевское значение) определяет, должно ли сохраняться данное свойство компонента при сохранении формы (true – должно, false – не должно) в файле с расширением .dfm. Последний спецификатор default определяет стандартное значение свойства, при совпадении значения поля с которым свойство не сохраняется.

Например:

```
private FValue:integer;
        procedure SetValue(AValue:integer);
        function StoreValue:boolean;
  published
        property Value:integer read FValue write SetValue
                              stored StoreValue default 10; . . .
```

Данное описание определяет свойство для внутреннего поля FValue некоторого класса. Чтение свойства выполняется напрямую из поля. Для записи в поле используется специальный метод SetValue. Сохранение в форме выполняется, если метод StoreValue возвращает true и значение отлично от 10.

Примечание. Принято называть поле класса, для которого определяется свойство, тем же именем, что и свойство, но с префиксом «F», а соответствующие параметры методов ввода и вывода с префиксом «A». Аналогично, имя метода чтения свойства желательно начинать с префикса «Get», метода записи – с «Set».

При необходимости в производном классе свойство можно переопределить, изменив доступность и/или спецификации. При этом можно указывать только изменяемые спецификации: остальные – наследуются.

В программе свойства используются как обычные поля, например:

A.Value :=n;

или

k:=A.Value;

соответственно при этом будут выполняться следующие операции:

A.SetValue(n);

и

k:=A.FValue;

Простые свойства целесообразно использовать:

– для ограничения доступа к полю (например, доступ только для чтения);

– при необходимости обеспечить при записи или чтении выполнение некоторых дополнительных действий (например, при записи в поле необходимо еще и фиксировать факт изменения значения).

Пример 5.3. Использование простых свойств (графический редактор «Окружности» – вариант 2). Продемонстрируем особенности программирования с использованием свойств на примере 5.1. В нем мы разрабатывали приложение, позволяющее рисовать на специальном поле окна приложения окружности.

Определим в классе два свойства Color и Radius. Изменение значений этих свойств свяжем с перерисовкой окружности. На рис. 5.9 представлена диаграмма класса TMyCircle для этого варианта.

Добавим еще одно изменение: поместим вызов процедуры рисования Draw в конструктор, так как конструирование объекта обычно связано с его рисованием. Текст модуля Circle имеет вид:

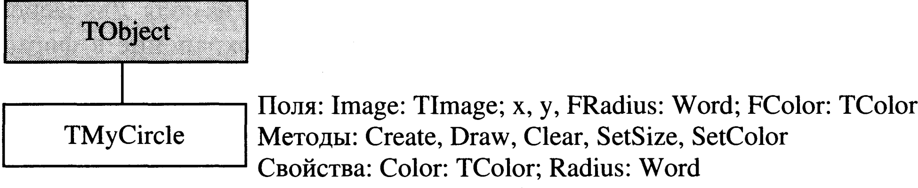


Рис. 5.9. Диаграмма класса TMyCircle

```
Unit Circle;
 Interface
  Uses  extctrls,Graphics;
  Type TMyCircle=class
         private    x,y,FRadius:Word;  FColor:TColor;  Image:TImage;
                    Procedure Clear;
                    Procedure Draw;
                    Procedure SetSize(ARadius:Word);
                    Procedure SetColor(AColor:TColor);
         public
              Constructor Create(aImage:TImage;ax,ay,ARadius:Word;
                                          AColor:TColor);
              property Radius:Word write SetSize;
              property Color:TColor write SetColor;
       end;
 Implementation
   Constructor TMyCircle.Create;
     Begin   inherited Create;
             Image:=aImage;   x:=ax;     y:=ay;
             FRadius:=ARadius;   FColor:=AColor;
             Draw;
     End;
   Procedure TMyCircle.Draw;
    Begin    Image.Canvas.Pen.Color:=FColor;
             Image.Canvas.Ellipse(x-FRadius, y-FRadius,
                                          x+FRadius, y+FRadius);
    End;
   Procedure TMyCircle.Clear;
    Var TempColor:TColor;
    Begin    TempColor:=FColor;
             FColor:=Image.Canvas.Brush.Color;
             Draw;
             FColor:=TempColor;
    End;
   Procedure TMyCircle.SetSize;
     Begin   Clear;    FRadius:=ARadius;  Draw;    end;
   Procedure TMyCircle.SetColor;
     Begin   Clear;    FColor:=AColor;    Draw;    end;
End.
```

При такой реализации класса вызов процедуры изменения размера и цвета рисунка будут программироваться как переназначение соответствующего свойства:

C.Radius:=strtoint(rEdit.Text);

или

C.Color:=ColorDialog.Color;

Примечание. Именно такой способ программирования лежит в основе всех стандартных компонент Delphi. Так, например, изменение свойства Visible, определенного для всех визуальных компонент, сопровождается в зависимости от ситуации появлением или исчезновением соответствующего компонента.

Анализ приведенного текста программы показывает, что использование простых свойств позволяет сократить количество компонентов класса, описанных в интерфейсной части (секция public), т.е. увеличивает степень инкапсуляции объектов.

Свойства-массивы. Описание свойств-массивов осуществляется следующим образом (квадратные скобки означают, что соответствующая спецификация может быть не указана):

property <имя> <список индексных параметров>:<тип>
[read <метод чтения>**] [write** <метод записи>**] [default];**

Список индексных параметров заключается в квадратные скобки. Это делает свойства-массивы внешне похожими на обычные массивы. Однако индексные параметры не являются индексами в привычном понимании этого термина. Значения индексных параметров, указанные при обращении к свойствам, передаются в методы чтения и записи в качестве обычных параметров. Соответственно тип индексных параметров может быть не только порядковым, но и вещественным или структурным.

В спецификациях доступа должны указываться методы, количество, тип и порядок следования параметров которых должны соответствовать списку индексных параметров. Параметр-значение в методе записи при этом указывается в конце списка.

Спецификация default используется для указания того, что из всех свойств объекта именно данное принимается по умолчанию (при обращении к объекту может быть опущено).

Например:

```
private function GetValMas(I:word;F:double):word;
        procedure SetValMas(I:word;F:double;AElement:word);
public  prorerty ValMas[I:word, F:double]:word
                 read GetValMas write SetValMas; default;
```

Приведенное описание означает, что для объекта определено некоторое свойство-массив, чтение и запись элементов которого выполняются

специальными методами. Причем при обращении к данному свойству имя свойства можно не указывать, так как оно объявлено используемым по умолчанию, например:

MyObject[k+2,s] :=1;

эквивалентно

MyObject.ValMas[k+2,s]:=1;

Свойства-массивы можно использовать для работы со сложными структурами, состоящими из однотипных элементов, например, динамическими массивами (так обычно называют массивы, размещенные в динамической памяти).

Пример 5.4. Использование свойств-массивов (класс «Динамический массив» – вариант 1). Пусть требуется разработать класс для реализации динамического массива с произвольным количеством элементов-байт. Максимальный размер массива необходимо определять при его создании. В процессе работы должен отслеживаться реальный размер массива.

Предусмотреть возможность выполнения следующих операций:

– ввод массива из элемента TSringGrid и вывод в такой же элемент;

– модификацию указанного элемента;

– вставку элемента на указанное место;

– удаление элемента с заданным индексом.

Кроме того, предусмотреть возможность обращения к любому элементу массива по индексу и обеспечить контроль правильности выполнения указанных операций.

```
Unit MasByte;
 Interface
   Uses SysUtils,Dialogs, Grids;
   Type
     TMas=array[1..255] of byte;
     TMasByte = class(TObject)      {класс «Динамический массив»}
       private
         ptr_an: ^TMas;      {указатель на массив}
         len:byte;           {максимальная длина массива}
         Procedure SetEl(Ind:byte;m:byte); {процедура записи элемента}
         Function GetEl(Ind:byte):byte;     {функция чтения элемента}
       public
         n:Byte;              {реальный размер массива}
         Constructor Create(an:byte);
         Destructor Destroy;override;
```

```
      Property Mas[Ind: byte]:byte read GetEl write SetEl;default;
      Procedure Modify(Ind:byte; Value:byte); {модификация элемента}
      Procedure Insert(Ind:byte; Value:byte);  {вставка элемента}
      Function Delete(Ind:byte):byte;          {удаление элемента}
      Function InputMas(Grid:TStringGrid;I,J:integer):boolean; {ввод}
      Procedure OutputMas(Grid:TStringGrid;I,J:integer);      {вывод}
    end;
Implementation
 Constructor TMasByte.Create;
   Begin
     inherited Create;
     GetMem(ptr_an,an);      len:=an;     { n:=0; указывать не надо}
   End;
 Destructor TMasByte.Destroy;
   Begin
     FreeMem(ptr_an);
     inherited Destroy;
   End;
 Procedure TMasByte.SetEl(Ind:byte;m:byte);
  Begin
    if Ind<=len then
      if Ind<=n then ptr_an^[Ind]:=m
      else
        MessageDlg('В массиве нет'+inttostr(Ind)+
                              '-го элемента.',mtError,[mbOk],0)
    else
       MessageDlg('В массиве можно разместить только'+
                        inttostr(Len)+' элементов.',mtError,[mbOk],0);
  End;
Function TMasByte.GetEl(Ind:byte):byte;
 Begin
   If  Ind<=n  then Result:=ptr_an^[Ind]
   else
      MessageDlg(' В массиве нет '+inttostr(Ind)+
                              '-го элемента.',mtError,[mbOk],0);
 End;
Function TMasByte.InputMas(Grid:TStringGrid;I,J:integer):boolean;
 Var k:byte; x,er_code:integer;
 Begin
    with Grid do
      begin
        k:=0;
       Result:=true;
```

```
          while (Cells[k+I,J]<>'')and Result do
            begin
              Val(Cells[k+I,J],x,er_code);
              if er_code=0 then
                if x<=255 then Insert(k+1,x)
                else
                 begin
                   MessageDlg(' Значение не может превышать 255.',
                                          mtError,[mbOk],0);
                   Result:=false;
                   Exit;
                 end
              else
                begin
                  MessageDlg('В строке обнаружены недопустимые
                                    символы.',mtError,[mbOk],0);
                 Result:=false;
                  Exit;
                end;
              k:=k+1;
            end;
          OutputMas(Grid,I,J);
        end;
    End;
Procedure TMasByte.OutputMas(Grid:TStringGrid;I,J:integer);
   Var k:byte;
   Begin
     with Grid do
        begin
          if n+I>ColCount then ColCount:=n+I;
          for k:=0 to ColCount-1 do
            if k<n then Cells[I+k,J]:=inttostr(mas[k+1])
            else Cells[I+k,J]:='';
        end;
   End;
Procedure TMasByte.Modify;
      Begin    Mas[Ind]:=Value;    End;
Procedure TMasByte.Insert;
   Var i:integer;
   Begin
      n:=n+1;   for i:=n-1 downto Ind do Mas[i+1]:=Mas[i];
      Mas[Ind]:=Value;
   End;
```

```
Function TMasByte.Delete;
   Var i:integer;
   Begin
     Result:=Mas[Ind];
     for i:=Ind+1 to n do Mas[i-1]:=Mas[i];  n:=n-1;
   End;
End.
```

Для работы с динамическим массивом необходимо создать соответствующий объект:

A:=TMasByte.Create(k);

после чего для обращения к элементам достаточно указать имя объекта и индекс, начиная с 1:

A[6]:=7;

Приведенный пример демонстрирует, что свойства-массивы создают иллюзию работы с массивом, позволяя не вникать в особенности реализации конкретной структуры.

Индексируемые свойства (свойства со спецификацией index). Описание индексируемых свойств выполняется следующим образом:

property <имя> : <тип> **index** <константа>
read <метод чтения> **write** <метод записи>;

где <константа> – целое число типа ShortInt.

Обычно одновременно описываются несколько свойств с различными значениями индексов, но одними и теми же методами чтения/записи. Методы чтения/записи должны содержать параметр, через который метод получит конкретное значение индекса. Причем в методе чтения данный параметр должен быть последним, а в методе записи – предпоследним, так как последним в нем описывается параметр, определяющий новое значение свойства. Значение параметра-индекса затем используется для определения поля, соответствующего свойству. Таким полем может быть, например, элемент массива:

```
Type CMas=class
     private
       FMasEl:array[1:3] of word;            {поле-массив}
       function GetEl(Ind:byte):word;        {метод чтения}
      procedure SetEl(Ind:byte;AElement);  {метод записи}
     public
```

```
        property Element1:word index 1 read GetEl write SetEl;
        property Element2:word index 2 read GetEl write SetEl;
        property Element3:word index 3 read GetEl write SetEl;
    end;
Function CMas. GetEl;
  Begin    Result:=FMasEl[Ind]; End;
Procedure CMas. SetEl;
  Begin FMasEl[Ind]:=AElement; End;
```

Свойства в этом случае использованы для обращения к элементам массива.

Процедурные свойства описывают интерфейс к полям, содержащим указатели на методы. Они в основном применяются для подключения методов обработки событий и будут рассмотрены в разделе 5.6.

5.4. Метаклассы

Delphi содержит достаточно богатый арсенал средств создания и обработки полиморфных объектов. Так в дополнение к заложенной еще в Borland Pascal 7.0 совместимости указателей на базовый и производные классы имеется возможность определения специальных переменных (*метаклассов*), значением которых являются классы. Имеются также специальные операции проверки принадлежности объекта некоторой иерархии классов (*is*) и уточнения класса заданного объекта (*as*).

Ссылка на класс (метакласс). В тех случаях, когда тип создаваемого объекта или требуемого метода класса не известен на этапе компиляции, обращение к соответствующему классу можно организовать с использованием специальной переменной типа *ссылка на класс* или *метакласс*. Объявление такой ссылки выполняется следующим образом:

```
Type <имя типа ссылки>=class of <базовый класс ссылки>;
Var  <имя переменной>:<имя типа ссылки>;
```

Значением переменной данного типа может служить имя базового класса ссылки или класса, производного от него.

В Delphi Pascal стандартно определена ссылка типа TClass, базовым классом которой является класс TObject. Переменной типа TClass можно присвоить имя любого класса Delphi, так как все они наследуются от класса TObject:

```
Type TClass=class of TObject;
Var c:TClass; d:TObject;. . .
     c:=TButton;
     d:=c.Create(. . .). . .
```

Операция is. Эта операция используется в тех случаях, когда возникает необходимость проверки принадлежности объекта некоторому классу или его потомкам:

<имя объекта> **is** <имя класса>

Операция возвращает true, если объект принадлежит классу, и false – в противном случае.

Операция as. Эта операция используется в тех случаях, когда тип объекта может отличаться от типа используемой переменной для обеспечения доступа к «невидимым» (см. рис. 1.25) полям и методам:

<имя объекта> **as** <имя класса>

В отличие от простого переопределения типа, использующего имя типа в качестве имени функции преобразования, операция as осуществляет проверку возможности преобразования типа, сравнивая тип реального объекта и тип указанного класса.

Методы класса. В Delphi существует возможность объявления методов, которые могут быть вызваны с указанием имени класса вместо имени объекта. Такие методы не получают указателя на поля объекта Self и называются *методами класса.*

Методы класса описываются со спецификатором *class*:

```
Type MyClass=class(TObject)
        public
          class procedure <имя метода>(<список параметров>);
      . . .
      end;
```

Метод класса не должен прямо или косвенно (через вызов других методов) использовать поля и свойства объекта, к которым у него нет доступа. Как правило, эти методы оперируют методами класса, объявленными в TObject, посредством которых можно получить информацию о классе, хранящуюся в ТВМ (см. далее).

Механизм определения типов на этапе выполнения программы. Все перечисленные выше возможности непосредственно обеспечиваются реализованным в Delphi механизмом определения типов на этапе выполнения.

Программа на Delphi Pascal содержит специальные таблицы, в которых хранится информация об используемых типах данных. Эта информация (RTTI – *Run Time Type Information* – «информация о типе времени выполнения») доступна во время выполнения программы. Такая информация хранится для типов, которые можно использовать в опубликованных свойствах (для целых и вещественных чисел, символов, перечислений, строк, множеств, классов,

методов и вариантных типов). Для каждого из указанных типов хранится специфическая информация, например, для целых – объем памяти, наличие знака, максимальное и минимальное значение, а для метода – вид (процедура или функция), количество параметров и их описание. RTTI информация используется самой средой Delphi при работе с компонентами в процессе конструирования приложения, но некоторые данные могут эффективно использоваться при разработке новых типов.

Примечание. Доступ к RTTI осуществляется с использованием специальной функции TypeInfo, в качестве параметра которой передается идентификатор типа из перечисленных выше. Для заданного типа TypeInfo возвращает указатель типа Pointer на таблицу RTTI данного типа. Эта таблица представляет собой запись типа TTypeInfo, описанного в модуле TypInfo, которая содержит имя типа и код его типа. Для получения конкретной информации о типе адрес этой таблицы следует передать в качестве параметра функции GetTypeData, которая вернет вариантную запись типа TTypeData, содержащую необходимую информацию.

Наибольший интерес представляет RTTI при работе с классами. Помимо стандартной RTTI классы сопровождаются дополнительной информацией, хранящейся в ТВМ.

ТВМ класса создается в процессе компиляции программы и существует даже тогда, когда не создано ни одного объекта данного класса. Все созданные для данного класса объекты совместно используют одну и ту же ТВМ – указатель на ТВМ класса содержится в первом автоматически формируемом поле объекта.

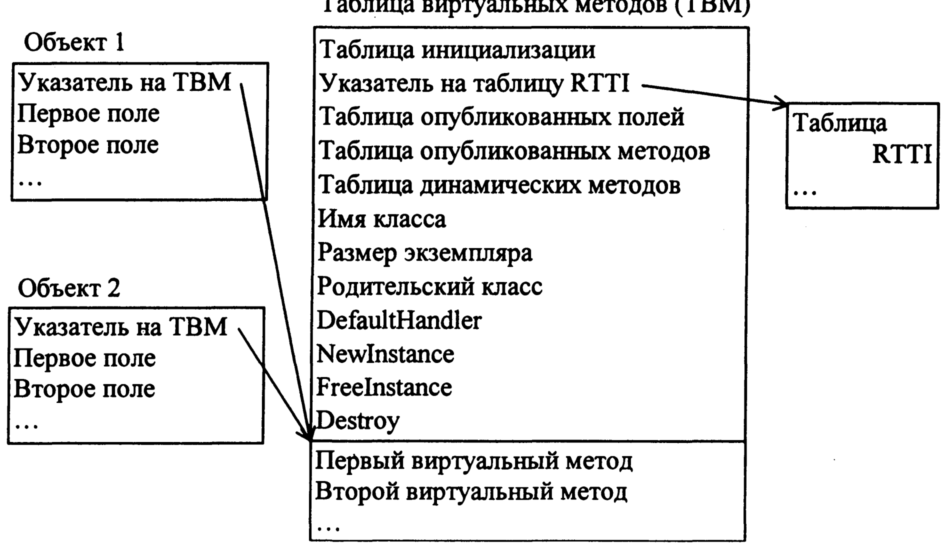


Рис. 5.10. Структура ТВМ

Перед адресом первого виртуального метода ТВМ содержит целый ряд указателей на различную информацию о классе (рис. 5.10), которая может потребоваться разработчику.

Вся эта информация, как и ТВМ, является общей для класса. В различных версиях Delphi она может различаться, так как совершенствование библиотек и средств, положенных в основу среды, требует хранения все большего количества информации о классе. В табл. 5.2 приведена структура ТВМ для Delphi 3.

Таблица 5.2. **Структура ТВМ**

Смещение	Содержимое
- 64	Двойной указатель на ТВМ
- 60	Двойной указатель на таблицу интерфейсов
- 56	Двойной указатель на таблицу информации OLE
- 52	Двойной указатель на таблицу инициализации
- 48	Двойной указатель на таблицу типов
- 44	Двойной указатель на таблицу определения полей
- 40	Двойной указатель на таблицу определения методов
- 36	Двойной указатель на таблицу динамических методов
- 32	Двойной указатель на строку, содержащую имя класса
- 28	Размер объекта в байтах (тип Cardinal)
- 24	Двойной указатель на класс-предок
- 20	Двойной указатель на метод SafecallException
- 16	Указатель на метод DefaultHandler
- 12	Указатель на метод NewInstance
- 8	Указатель на метод FreeInstance
- 4	Указатель на метод Destroy
0	Указатель на первый виртуальный метод

Эти данные можно использовать в программе непосредственно, так, например, для получения имени класса можно указать:

ShortString(Pointer(Pointer(Integer(TEdit) - 32)^)^).

В классе TObject определены методы класса, обеспечивающие доступ к некоторой информации ТВМ:

1) **class function ClassName: ShortString;** – возвращает имя класса;

2) **class function ClassNameIs(const Name: string): Boolean;** – возвращает true, если имя класса совпадает с указанным в параметре;

3) **class function ClassParent: TClass;** – возвращает объектную ссылку на предка;

4) **class function ClassInfo: Pointer;** – возвращает указатель на таблицу RTTI класса;

5) **class function InstanceSize: Longint;** – возвращает размер экземпляра объекта;

6) **class function InheritsFrom(AClass: TClass): Boolean;** – проверяет, наследуется ли данный класс от указанного;

7) **class function MethodAddress(const Name: ShortString): Pointer**; – возвращает адрес метода по его имени;

8) **class function MethodName(Address: Pointer): ShortString;** – возвращает имя метода по его адресу.

Поскольку все объекты содержат один и тот же адрес ТВМ, для проверки принадлежности объектов одному классу достаточно сравнить адреса ТВМ.

Проверка принадлежности объекта иерархии классов реализована как поиск среди ТВМ заданного и производных классов таблицы, адрес которой совпадает с указанным в объекте. Возможность преобразования типа для объекта также проверяется по соответствию адресов ТВМ.

Пример 5.5. Контейнер «Двусвязный линейный список». В качестве примера использования RTTI разработаем контейнерный класс, который можно будет использовать для хранения в виде списка объектов любых типов, например, чисел и символьных строк.

Для организации контейнера используются два класса: класс TSpisok и класс TElement. Класс TSpisok организует список из объектов TElement, включающих только два поля suc и pre, используемые для хранения адресов следующего и предыдущего элементов двусвязного списка.

Диаграммы контейнерного класса и его элемента приведены на рис. 5.11.

Модуль Spisok, содержащий описание классов TElement и TSpisok, выглядит следующим образом:

```
Unit Spisok;
Interface
 Type TElement=class(TObject)
        public     pre,suc:TElement;
    end;
```

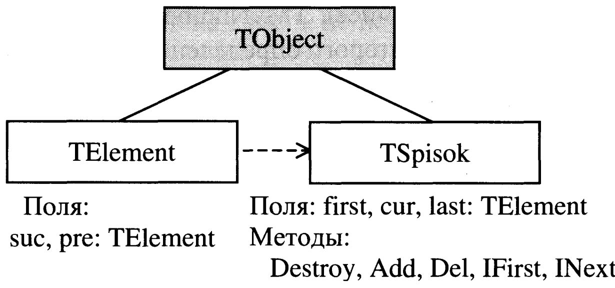


Рис. 5.11. Диаграмма классов TSpisok и TElement

```
    TSpisok=class(TObject)
      private     first,last,cur: TElement;
      public      Destructor Destroy;override;
                  Procedure Add(e:TElement); {добавление}
                  Function Del:TElement;      {удаление}
                  Function IFirst:TElement;   {первый элемент}
                  Function INext:TElement;    {следующий элемент}
    end;
Implementation
 Destructor TSpisok.Destroy;
  Var v:TElement;
  Begin     v:=Del;
            while v<>nil do  begin  v.Destroy;  v:=Del;   end;
            inherited Destroy;
  End;
 Procedure TSpisok.Add;
  Begin    if first=nil then   begin first:=e; last:=e;  end
           else  begin e.suc:=first; first.pre:=e; first:=e; end;
  End;
 Function TSpisok.Del;
  Begin Del:=last;
        if last<>nil then
          begin  last:=last.pre;  if last<>nil then last.suc:=nil;  end;
        if last=nil then first:=nil;
  End;
 Function TSpisok.IFirst;
     Begin   cur:=first;    Result:=cur;   End;
 Function TSpisok.INext;
     Begin   cur:=cur.suc;    Result:=cur;   End;
End.
```

При создании реального приложения на базе контейнерного класса и его элемента строятся классы, реализующие конкретные особенности решаемой задачи (в данном случае класс чисел TMyNumber, класс строк TMyString и класс список TMySpisok, для которого определено суммирование элементов-чисел). Диаграммы этих классов представлены на рис. 5.12.

Текст модуля MySpisok:

```
Unit MySpisok;
 Interface
  Uses Spisok;
  Type TMyNumber=class(TElement)
       private      FMyNum:integer;
```

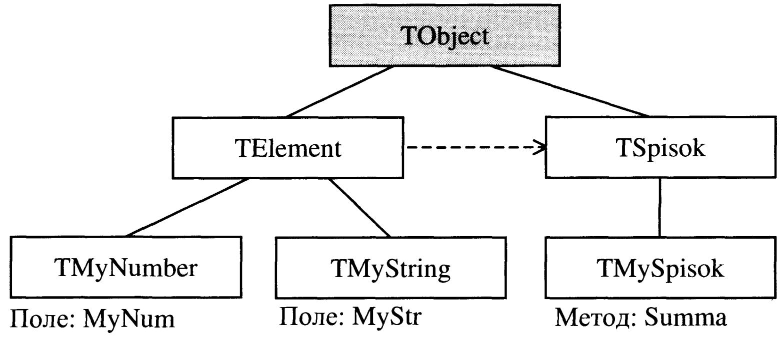


Рис. 5.12. Создание конкретного списка на базе класса TSpisok

```
public      Constructor Create(aMyNum:integer);
             property MyNum:integer
                            read FMyNum write FMyNum;
end;
TMyString=class(TElement)
  private     FMyStr:string;
  public      Constructor Create(aMyStr:string);
              property MyStr:string read FMyStr Write FMyStr;
end;
TMySpisok=class(TSpisok)
  public   Function Summa:integer;
end;
Implementation
 Constructor TMyNumber Create;
   Begin      inherited Create;
          FMyNum:=aMyNum;    End;
 Constructor TMyString Create;
   Begin      inherited Create;      FMyStr:=aMyStr;    End;
 Function TMySpisok.Summa;
 Var c:TElement;
 Begin Result:=0;    c:=IFirst;   {взять первый элемент списка}
    while c<>nil do       {пока список не исчерпан}
     begin
      if c is TMyNumber then  {если c – объект-число, то }
          Result:=Result+(c as TMyNumber).MyNum; {преобразовать, так
                     как иначе поле MyNum – невидимо, и суммировать }
      c:=INext;        {переходим к следующему числу}
     end;
 End;
End.
```

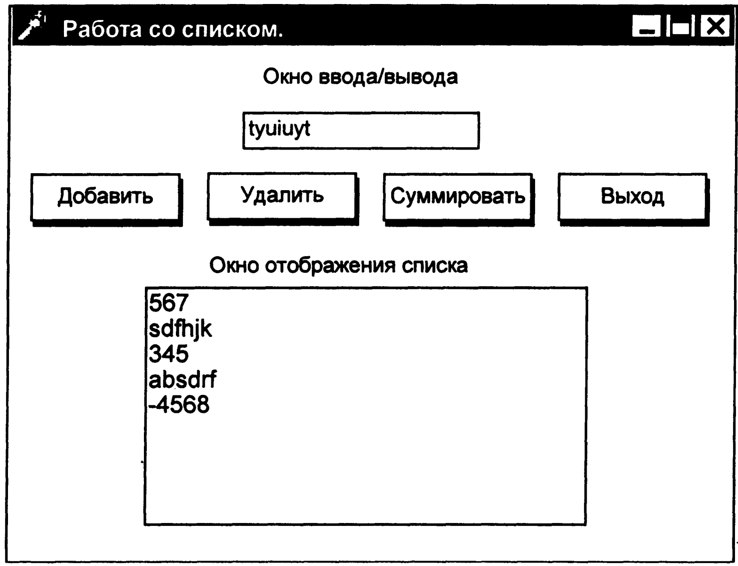


Рис. 5.13. Вид главного окна приложения «Список»

Наибольший интерес в данном модуле вызывает реализация функции суммирования, которая использует итераторы, предлагаемые в контейнерном классе, для организации обработки всех элементов списка.

В этой же функции возникает необходимость определения принадлежности объекта некоторому классу и преобразования типа объекта для обеспечения видимости полей. Поскольку список может состоять из объектов разных классов, то для работы с ними используется указатель на базовый тип TElement. Конкретные объекты, включенные в список, содержат дополнительные поля, которые естественно не видны указателю базового типа.

Вид окна программы тестирования приведен на рис. 5.13. При нажатии на соответствующую кнопку должна быть выполнена указанная на ней операция. После выполнения каждой операции окно отображения списка должно корректироваться.

```
Unit Main;
 Interface
   Uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
                         logs, StdCtrls;
   Type
    TMainForm = class(TForm)
        AddButton, DelButton, SumButton, ExitButton: TButton;
        IOEdit: TEdit; IOLabel, ListLabel: TLabel;
        ListBox1: TListBox;
        Procedure ExitButtonClick(Sender: TObject);
```

```
        Procedure SumButtonClick(Sender: TObject);
        Procedure AddButtonClick(Sender: TObject);
        Procedure IOEditKeyPress(Sender: TObject; var Key: Char);
        Procedure DelButtonClick(Sender: TObject);
        Procedure FormActivate(Sender: TObject);
    end;
  Var  MainForm: TMainForm;
Implementation
 Uses MySpisok,Spisok;
 Var S:TMySpisok;
  {$R *.DFM}
 Procedure TMainForm.FormActivate(Sender: TObject);
    Begin  IOLabel.Visible:=false;  IOEdit.Visible:=false;     End;
 Procedure TMainForm.AddButtonClick(Sender: TObject);
     Begin   IOLabel.Visible:=true;
             IOEdit.Visible:=true;
             IOEdit.SetFocus;
     End;
Procedure TMainForm.IOEditKeyPress(Sender: TObject;  var Key: Char);
 Var  I, Code: Integer; v:TElement;
 Begin
  if Key=#13 then
    begin
     Key:=#0;
     Val(IOEdit.Text, I, Code);
     Listbox1.Items.Add(IOEdit.Text);
      if Code = 0   then                        {если введено число, то}
          v:=TMyNumber.Create(I)                 {создать объект-число}
      else v:=TMyString Create(IOEdit.Text);   {иначе – создать строку}
      S.Add(v);                                 {добавить объект к списку}
     AddButton.SetFocus;
     IOLabel.Visible:=false;       IOEdit.Visible:=false;
   end;
  End;
Procedure TMainForm.DelButtonClick(Sender: TObject);
  Var v:TElement;
  Begin
   IOLabel.Visible:=true;      IOEdit.Visible:=true;
   v:=S.Del;                {удалить объект из списка}
   if v<>nil then           {если в списке есть объекты, то}
    begin
     ListBox1.Items.Delete(0);   {удалить отображение строки}
     if v is TMyNumber then      {если удаленный объект-число, то}
```

```
        IOEdit.Text:=inttoStr((v as TMyNumber).MyNum){вывести число}
      else IOEdit.Text:=(v as TMyString).MyString;  {иначе – вывести строку}
      v.Free;                                       {уничтожить объект}
      end
    else IOEdit.Text:='Список пуст.';
  End;
Procedure TMainForm.SumButtonClick(Sender: TObject);
  Begin
    IOLabel.Visible:=false;
    IOEdit.Visible:=false;
    Application.MessageBox(Pchar(IntToStr(S.Summa)), 'Сумма:',MB_OK);
  End;
Procedure TMainForm.ExitButtonClick(Sender: TObject);
Begin
Close;
End;
Initialization S:=TMySpisok.Create;      {создать объект-список}
Finalization   S.Destroy;               {уничтожить объект-список}
End.
```

5.5. Делегирование

В Delphi существует возможность объявления *указателей на методы*. Как уже упоминалось в разделе 5.1, в отличие от обычных подпрограмм (процедур и функций) методы в списке параметров содержат неявную ссылку на объект, которая обеспечивает методу доступ к полям и методам «своего» объекта. Соответственно, указатель на метод обеспечивает корректную передачу этой ссылки.

При объявлении типа «указатель на метод» используется специальный спецификатор of object:

Type <имя типа> = <заголовок метода> **of object;**

Например:

```
Type TMyMetod = function (ax:integer):word of object;
     TMyClass=class
       public  FMetod:TMyMetod;
     end;
```

Поле FMetod, описанное выше, имеет процедурный тип, его значением может быть имя метода данного или другого класса.

В Delphi доступ к полям, содержащим указатели на методы, обычно осуществляется через свойства:

```
Type TMyClass=class
       private FMetod:TMyMetod;
       public property Metod:TMyMetod read FMetod write FMetod;
     end;
```

Как уже упоминалось в разделе 5.4, такие свойства получили название *процедурных*. С использованием процедурных свойств в Delphi осуществляется *делегирование* (раздел 1.7), которое позволяет определять *различное поведение объектов одного класса.*

Так подключение обработчиков событий стандартных компонент представляет собой *статическое* делегирование, при котором поведение объекта определяется в процессе проектирования приложения.

При *динамическом* делегировании поведение объекта уточняется *во время выполнения программы*, что позволяет создавать объекты с динамически изменяющимся поведением.

Пример 5.6. Делегирование методов (графический редактор «Окружности и квадраты» – вариант 2). В качестве примера рассмотрим создание приложения, которое в зависимости от положения радиокнопки по нажатию левой клавиши мыши рисует в специальном окне окружность или квадрат. Аналогичный пример уже рассматривался в разделе 5.2, но там нужный эффект достигался за счет переопределения методов. Классы TCircle и TSquare включали свои методы рисования Draw. Смена типа фигуры реализовывалась за счет создания объекта другого класса. Изменить тип уже созданного объекта в рассмотренном там варианте решения нельзя.

Использование делегирования позволяет создать объект, который при изменении положения радиокнопки будет изменять тип и соответственно вид уже нарисованной фигуры (рис. 5.14).

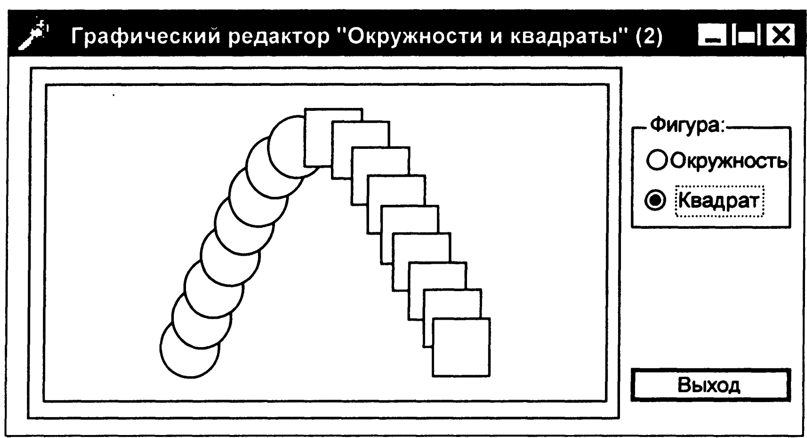


Рис. 5.14. Вид главного окна приложения «Окружности и квадраты»

Вариант 1. Простейший вариант создания такого объекта заключается в том, чтобы включить в описание класса TFigure обе процедуры рисования фигур DrawCircle (рисование окружности) и DrawSquare (рисование квадрата), осуществляя вызов процедуры рисования через свойство Draw – указатель на метод рисования. Для этого придется определить тип указателя на метод без параметров TDProc.

```
unit Figure;
interface
Uses  extctrls,Graphics;
Type TDProc=procedure of object;
   TMyFigure=class
    private
     x,y,r:word;
     Image:TImage;
     FDraw:TDProc;  {поле свойства Draw}
    public
     Constructor Create(aImage:TImage;ax,ay,ar:Word); {конструктор}
     property Draw:TDProc read FDraw write FDraw; {процедурное
                                                      свойство}
     Procedure Clear;            {стирание фигуры}
     Procedure DrawCircle;       {рисование окружности}
     Procedure DrawSquare;       {рисование квадрата}
    end;
Implementation
Constructor TMyFigure.Create;
  Begin
    inherited Create;   Image:=aImage;
    x:=ax;   y:=ay;   r:=ar;
  End;
Procedure TMyFigure.Clear;
 Begin    Image.Canvas.Pen.Color:=clWhite;
          Draw;  {вызов метода по адресу, указанному в свойстве}
         Image.Canvas.Pen.Color:=clBlack;
 End;
Procedure TMyFigure.DrawCircul;
   Begin    Image.Canvas.Ellipse(x-r,y-r,x+r,y+r);  End;
Procedure TMyFigure.DrawSquare;
   Begin    Image.Canvas.Rectangle(x-r,y-r,x+r,y+r);  End;
end.
```

Вид выводимой фигуры определяется «нажатой» радиокнопкой:

```
     case RadioGroup.ItemIndex of
```

```
          0: C.Draw:=C.DrawCircle;
          1: C.Draw:=C.DrawSquare;
       end;
```

Переопределение свойства Draw (указателя на метод рисования) выполняется в обработчике события **RadioGroupClick** (щелчок мышью на радиокнопках):

```
Procedure TMainForm.RadioGroupClick(Sender: TObject);
 Begin  if C<>nil then                         {если объект создан, то}
        begin
          C.Clear;                             {стереть изображение}
          case RadioGroup.ItemIndex of  {в зависимости от выбора}
             0: C.Draw:=C.DrawCircle; { рисование окружности}
             1: C.Draw:=C.DrawSquare; { рисование квадрата}
         end;
        C.Draw;                   {нарисовать фигуру}
      end;
 End;
```

В а р и а н т 2. Более сложный вариант делегирования – подключение методов, определенных в других классах.

Определим специальный класс TDraw, который будет содержать методы рисования. Необходимые параметры эти методы будут получать через список аргументов, которые сгруппированы в запись типа TParam.

```
Unit Figure;
 Interface
  Uses  extctrls,Graphics;
  Type TParam=record
          x,y,r:Word;
          Color:TColor;
          Image:TImage;
        end;
    TDProc=Procedure(AParam:TParam) of object;
    TMyFigure=class
       private     FDraw:TDProc;
       public
          Param:TParam;
          Procedure Clear;
          Constructor Create(aImage:TImage;ax,ay,ar:Word;aColor:TColor);
          property Draw:TDProc read FDraw write FDraw;
     end;
    TDraw=class
```

```
      public
        Procedure DrawCircle(AParam:TParam);
        Procedure DrawSquare(AParam:TParam);
      end;
Implementation
 Constructor TMyFigure.Create;
   begin
     inherited Create;
     Param.Image:=aImage; Param.Color:=aColor;
     Param.x:=ax;   Param.y:=ay;   Param.r:=ar;
   end;
 Procedure TMyFigure.Clear;
  Var TempColor:TColor;
  Begin
    TempColor:=Param.Color;
    Param.Color:=Param.Image.Canvas.Brush.Color;
    Draw(Param);                 {вызов делегированного метода}
    Param.Color:=TempColor;
  End;
Procedure TDraw.DrawCircle;
  Begin
    AParam.Image.Canvas.Pen.Color:=AParam.Color;
    AParam.Image.Canvas.Ellipse(AParam.x-AParam.r, AParam.y-AParam.r,
                                AParam.x+AParam.r,AParam.y+AParam.r);
  End;
Procedure TDraw.DrawSquare;
  Begin
    AParam.Image.Canvas.Pen.Color:=AParam.Color;
    AParam.Image.Canvas.Rectangle(AParam.x-AParam.r,
          AParam.y-AParam.r, AParam.x+AParam.r, AParam.y+AParam.r);
  End;
End.
```

Переопределение свойства Draw в обработчике события **RadioGroupClick** теперь будет выполняться следующим образом:

```
Procedure TMainForm.RadioGroupClick(Sender: TObject);
 Begin
   if C<>nil then
      begin   C.Clear;
              case RadioGroup.ItemIndex of
              0: C.Draw:=G.DrawCircle;  {делегирование метода}
              1: C.Draw:=G.DrawSquare;  {делегирование метода}
              end;
```

```
            C.Draw(C.Param);     {вызов процедуры рисования}
      end;
  End;
```

Объект G класса TDraw может быть создан в секции инициализации модуля и уничтожен – в секции завершения:

```
initialization G:=TDraw.Create;
finalization  G.Free;
```

5.6. Библиотека стандартных классов Delphi

Библиотека стандартных классов Delphi VCL содержит сотни классов, на базе которых разработчик может создавать собственные приложения Windows.

На рис. 5.15 показана иерархия основных базовых классов библиотеки VCL, на базе которых созданы все остальные классы Delphi.

Как уже упоминалось, все компоненты библиотеки наследуются от класса TObject, который содержит целый ряд специальных методов, существенно снижающих сложность программирования в Delphi:

```
type   TObject = class
         constructor Create;             {конструктор}
         destructor Destroy; virtual;    {деструктор}
```

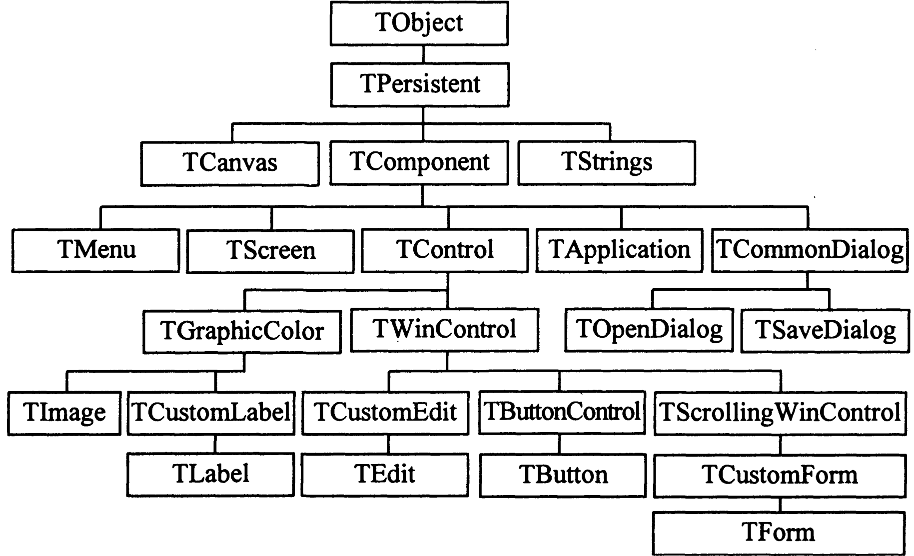


Рис. 5.15. Иерархия классов основных компонент библиотеки VCL

```
procedure Free;            {уничтожить, если элемент был создан}
class Function InitInstance(Instance: Pointer): TObject; {инициализирует
                 память при создании объекта}
class Function NewInstance: TObject; virtual; {выделяет пямять для
                                     размещения объекта}
Procedure CleanupInstance; {осуществляет корректное завершение
          работы со строками и другими сложными структурами при
          уничтожении объекта}
Procedure FreeInstance; virtual; {освобождает память, выделенную
                             под размещение объекта}
Function ClassType: TClass; {возвращает класс объекта}
class Function ClassName: ShortString; {возвращает имя класса}
class Function ClassNameIs(const Name: string): Boolean; {проверяет
         принадлежность объекта указанному классу}
class Function ClassParent: TClass; {возвращает тип предка}
class Function ClassInfo: Pointer; {возвращает указатель на таблицу
                                      RTTI}
class Function InstanceSize: Longint; {возвращает размер объекта в
                                          байтах}
class Function InheritsFrom(AClass: TClass): Boolean; {проверяет
       принадлежность класса или объекта семейству указанного
       класса}
Procedure Dispatch(var Message); {посылает сообщение объекту}
class Function MethodAddress(const Name: ShortString): Pointer;
             {возвращает адрес опубликованного метода по имени}
class Function MethodName(Address: Pointer): ShortString;
          {возвращает имя опубликованного метода по его адресу}
Function FieldAddress(const Name: ShortString): Pointer;
          {возвращает адрес опубликованного поля по его имени}
Function GetInterface(const IID: TGUID; out Obj): Boolean; {проверяет
     соответствие указанного интерфейса класс}
class Function GetInterfaceEntry(const IID: TGUID):
          PInterfaceEntry; {возвращает указатель на структуру,
         содержащую описание специального интерфейса класса}
class Function GetInterfaceTable: PInterfaceTable; {возвращает
     указатель на структуру, содержащую описание интерфейса для
     класса}
Function SafeCallException(ExceptObject: TObject;
                            ExceptAddr: Pointer): Integer; virtual;
                          {метод поддержки исключений OLE}
Procedure DefaultHandler(var Message); virtual; {выполняет обработку
                          сообщения по умолчанию}
end;
```

От класса TObject наследуется родоначальник всех классов, которые могут иметь секцию published – класс *TPersistent.* Этот класс обеспечивает корректную работу Инспектора Объектов с опубликованными свойствами (сохранение их в файлах формы, инициализацию опубликованных свойств при компиляции программы и копирование полей одного объекта в другой).

От класса TPersistent наследуются классы TComponent (компоненты) – родоначальник всех классов-компонент и некоторые другие вспомогательные классы, например: TStrings (строки), TCollection (коллекции), TCanvas («холсты» – поля, на которых можно «рисовать»), TGraphicObject (графические объекты), TGraphic (графические элементы), TPicture (изображения).

От класса TComponent наследуются все *компоненты* приложения, в том числе и само приложение (класс TApplication). Особенность потомков класса TComponent заключается в том, что объекты-компоненты могут находиться между собой в отношении «основной – вспомогательный».

Основной компонент отвечает за управление памятью при размещении вспомогательных компонентов: выделение памяти при создании компонента и освобождение памяти при его уничтожении. Для всех компонент, размещенных в форме (TForm), основным компонентом является форма, а для самой формы основным компонентом является приложение (TApplication).

Реализация отношения «основной – вспомогательный» в классе TComponent осуществляется с использованием следующих свойств:

1) Owner – должно содержать указатель на основной компонент для текущего компонента (оно инициализируется автоматически при помещении компонента в форму);

2) ComponentIndex – содержит номер текущего компонента в массиве Components (начиная с 0) основного компонента; определяет порядок создания и изображения вспомогательных компонентов;

3) Components[Index] – свойство-массив типа TComponent, содержит указатели на все вспомогательные компоненты текущего компонента;

4) ComponentCount – содержит количество вспомогательных компонент текущего компонента.

Таким образом, при необходимости можно просмотреть свойство-массив Components размером ComponentCount и найти нужные вспомогательные компоненты.

Особое место среди потомков TComponent занимает класс TControl, от которого наследуются все *элементы управления*, размещаемые в окне формы. Соответственно, этот класс содержит свойства, определяющие расположение элементов управления относительно формы на экране.

Считается, что любой элемент управления занимает прямоугольник внутри окна формы. Положение этого прямоугольника определяется относительно окна формы координатами верхнего левого и нижнего правого углов.

Класс TControl определяет методы, обрабатывающие сообщения мыши

и обеспечивающие генерацию соответствующих событий (обычное, двойное нажатие, движение с нажатыми клавишами и т.п.).

От класса TControl наследуются классы TWinControl – оконные элементы управления и TGraphicControl – графические элементы управления.

Оконные элементы управления имеют собственную *функцию окна* и, соответственно, могут получать сообщения Windows (TEdit, TMemo, TListBox), в том числе сообщения от клавиатуры, т.е. могут получать *фокус ввода.* Соответственно, класс TWinControl включает методы обработки сообщений клавиатуры, которые формируют события клавиатуры.

Графические элементы управления порождаются от TGraphicControl и не могут обрабатывать ввод с клавиатуры (TLabel, TImage, TBevel). Они в основном используются для отображения информации.

Класс TWinControl устанавливает между оконными элементами управления отношение *«старший – младший»*. Это отношение определяет подчиненность изображений оконных элементов управления на экране. Например, если форма становится невидимой, то невидимыми становятся все ее младшие элементы управления (метки, кнопки и т.д.).

То же отношение позволяет управлять единообразием изображения старших и младших компонент (шрифт, цвет и т.п.).

Отношения «основной – вспомогательный» и «старший – младший» не следует путать. Рассмотрим пример, демонстрирующий различие между этими отношениями.

Например, определим форму, которая включает несколько визуальных и невизуальных компонентов (рис. 5. 16).

Кроме этого, пусть соответствующий класс также содержит объектное поле text типа TStrings:

Type
TMainForm = class(TForm)
GroupBox: TGroupBox;   CheckBox1: TCheckBox;

Рис. 5.16. Форма с несколькими компонентами (на этапе проектирования)

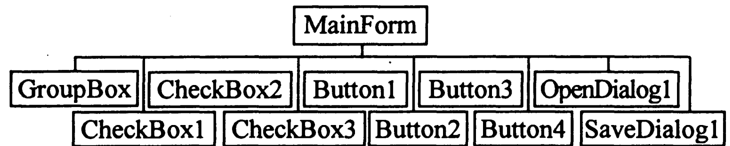


Рис. 5.17. Отношение «основной – вспомогательный»

```
CheckBox2: TCheckBox;   CheckBox3: TCheckBox;
  Button1: TButton;   Button2: TButton;
  Button3: TButton;   Button4: TButton;
  OpenDialog1: TOpenDialog;   SaveDialog1: TSaveDialog;
 private
  text:TStrings; . . .
 end;
```

Список компонентов объекта MainForm будет включать все поля, кроме text, так как последнее не является потомком класса TComponent. Объект MainForm при этом является основным, а его компоненты по отношению к нему – вспомогательными (рис.5.17). При конструировании формы автоматически будут конструироваться все вспомогательные компоненты этой формы. О конструировании поля text разработчик должен позаботиться сам.

Список младших по отношению к объекту MainForm оконных управляющих элементов выглядит существенно короче: GroupBox, Button1, Button2, Button3, Button4, так как компоненты OpenDialog1 и SaveDialog1 не являются управляющими элементами (потомками TControl). Элемент GroupBox является старшим по отношению к CheckBox1, CheckBox2, CheckBox3 (рис. 5.18). Следовательно, при сокрытии формы будут невидимы все визуальные элементы управления, а при сокрытии элемента GroupBox станут невидимыми и CheckBox1, CheckBox2, CheckBox3.

Для определения отношения «старший – младший» класс TWinControl включает следующие свойства:

1) **Parent** – содержит указатель на старший элемент управления;

2) **ControlIndex** – содержит номер текущего элемента управления в массиве Controls (начиная с 0) старшего элемента управления, определяющий порядок передачи фокуса ввода среди младших элементов управления;

3) **Controls[Index]** – свойство-массив типа TControl, содержит указатели на все младшие элементы управления по отношению с текущему старшему;

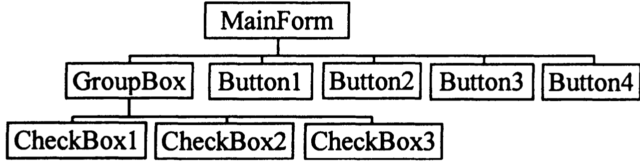


Рис. 5.18. Отношение «старший – младший»

4) **ControlCount** – содержит количество младших элементов управления для текущего старшего элемента.

Для управления единообразием изображений используются следующие свойства:

1) **ParentColor:boolean** – определяет, будут ли использоваться цвета родительского элемента или элемент будет сам устанавливать цвет;

2) **ParentFont:boolean** – определяет, будут ли использоваться параметры шрифта родительского элемента или должны использоваться собственные установки;

3) **ParentShowHint:boolean** – определяет, будет ли использоваться свойство ShowHint родительского элемента или собственное.

Кроме этого, существуют специальные методы, позволяющие управлять процессом передачи фокуса ввода:

1) **SetFocus** – позволяет установить фокус ввода на нужный оконный элемент управления;

2) **FindNextControl** – возвращает следующий элемент в цепочке, установленной TabOrder;

и свойства

3) **Enabled** – определяет, может ли данный элемент принимать фокус ввода;

4) **TabOrder** – определяет порядок передачи фокуса ввода при нажатии клавиши Tab;

5) **ActiveControl** родительского элемента – содержит адрес элемента управления, на который в настоящий момент установлен фокус ввода.

Используя отношение «старший – младший», можно определить однотипное поведение оконных элементов управления.

Пример 5.7. Использование отношения «старший – младший» (приложение «Определение вида четырехугольника»). Пусть необходимо разработать приложение, которое должно определять внешний вид четырехугольника по заданным координатам вершин.

Для ввода координат вершин понадобится 8 элементов TEdit (рис. 5.19).

При создании компонента TEdit было предусмотрено, что перемещение курсора на следующий компонент окна выполняется при нажатии клавиши Tab, хотя удобнее было бы, если такое переключение происходило при нажатии клавиши Enter. Для того чтобы изменить переключение по клавише Tab на переключение по Enter, необходимо для каждого элемента TEdit написать обработчик события KeyPress. Эти обработчики должны выполнять одинаковые действия для каждого TEdit. Аналогично, однотипные действия над этими компонентами должны выполняться и при активации формы.

Для уменьшения размера программы, например, за счет выполнения однотипных действий в цикле, воспользуемся свойствами, определяющими отношение «старший – младший». Конкретно, для описанного ниже класса

Рис. 5.19. Окно приложения в момент выдачи сообщения об ошибке

TMainForm выполним настройку формы (событие **FormActivate** – активация формы), универсальную обработку нажатия клавиши Enter (событие **AllEditKeyPress** – ввод Enter для любого TEdit) и ввод данных из всех 8 элементов TEdit (событие **ExButtonClick** – нажатие кнопки «Определить вид», невидимой на рисунке).

```
Type
 TMainForm = class(TForm)
  Bevel1: TBevel; InputLabel: TLabel;
  ALabel: TLabel; BLabel: TLabel;
  CLabel: TLabel; DLabel: TLabel;
  AxEdit: TEdit;  AyEdit: TEdit;
  BxEdit: TEdit;  ByEdit: TEdit;
  CxEdit: TEdit;  CyEdit: TEdit;
  DxEdit: TEdit;  DyEdit: TEdit;
  NextButton: TButton; ExitButton: TButton;ExButton: TButton;
  ResLabel: TLabel; ResultLabel: TLabel;
  x1Label: TLabel; y1Label: TLabel;
  x2Label: TLabel; y2Label: TLabel;
  procedure AllEditKeyPress(Sender: TObject; var Key: Char);
  procedure FormActivate(Sender: TObject);
  procedure ExButtonClick(Sender: TObject);
  procedure NextButtonClick(Sender: TObject);
  procedure ExitButtonClick(Sender: TObject);
 end;
```

```
. . .
Implementation
 Var K:array[1..8] of Double;
 Procedure TMainForm.FormActivate(Sender: TObject);
  Var i:integer;
   Begin
    for i:=0 to ControlCount-1 do        {для всех младших управляющих
                                                   элементов формы}
      if Controls[i] is TEdit then          {если элемент – TEdit, то}
       begin
        Controls[i].Enabled:=true;          {разрешить получение фокуса}
        (Controls[i] as TEdit).ReadOnly:=false;  {разрешить ввод}
      end;
   ExButton.Visible:=true;                  {показать кнопку «Выполнить»}
   NextButton.Enabled:=false;   {запретить передачу фокуса кнопке
                                                 «Выполнить»}
   ResLabel.Visible:=false;        {спрятать метку результата}
   ResultLabel.Visible:=false;    {спрятать поле результата}
   AxEdit.SetFocus; {установить фокус на первый TEdit (TabOrder=0)}
 End;
Procedure TMainForm.AllEditKeyPress(Sender: TObject; var Key: Char);
 Var Code:integer; S:string;
 Begin
   if Key=#13 then              {если нажата клавиша Enter, то}
     begin
       Key:=#0;                  {отменить сигнал ошибки – звонок}
       Val((ActiveControl as Tedit).Text,
                                  K[(ActiveControl as Tedit).Tag],Code);
          {преобразовать введенные данные в число и разместить его в
          массиве K с индексом из поля Tag данного элемента}
       if Code<>0 then   {если обнаружена ошибка преобразования, то}
         begin          {сформировать сообщение об ошибке}
          S:=chr(ord('A')+ActiveControl.TabOrder div 2);
          MessageDlg('Координаты '+S+
                                   'введены неверно.',mtError,[mbOk], 0);
          exit;
         end;
      FindNextControl(ActiveControl,true,false,false).SetFocus; {передать
                фокус следующему активному элементу}
    end
 End;
Procedure TMainForm.ExButtonClick(Sender: TObject);
 Var Code,i,j:integer;    PP:TEdit; S:string; . . .
```

```
Begin
  PP:=AxEdit;          {установить фокус на первый элемент (TabOrder = 0)}
  for i:=1 to 8 do      {для 8 последовательно расположенных элементов}
   begin
     Val(PP.Text, K [PP.Tag],Code); {выполнить преобразование}
       if Code<>0 then               {если обнаружена ошибка, то}
         begin                       {выдать сообщение об ошибке}
           S:=chr(ord('A')+PP.TabOrder div 2);
           PP.SetFocus;
           MessageDlg('Координаты '+S+' введены неверно.',
                                                   mtError, [mbOk], 0);
           exit;
         end;
     if i<>8 then PP:=FindNextControl(PP,true,false,true) as TEdit; {передать
              фокус следующему активному элементу}
   end;
....
    {если логических ошибок не обнаружено, то запретить установку фокуса
     ввода на элементы ввода данных и объявить их «только для чтения»}
    PP:=AxEdit;
    for i:=1 to 8 do
     begin
       PP.Enabled:=false;
       PP.ReadOnly:=true;
       if i<>8 then PP:=FindNextControl(PP,true,false,true) as TEdit;
    end;
  . . .
End;     . . .
```

В программе использовано также свойство Tag. Это свойство класса TComponent может хранить любую информацию разработчика в виде 32-разрядного целого числа. В нашем случае оно хранит индекс вводимой координаты в массиве K, используемом для записи координат.

5.7. Создание и обработка сообщений и событий

Стандартная библиотека классов Delphi VCL предлагает разработчику достаточно большой набор сообщений и методов их разработки. Однако он имеет возможность добавить новое сообщение или переопределить методы обработки существующих сообщений.

При создании нового сообщения необходимо выполнить следующие действия:

1) описать тип сообщения;

2) объявить номер (или индекс) сообщения;

3) объявить метод обработки нового сообщения в классе, который должен его обрабатывать;

4) инициализировать (передать) сообщение.

Сообщения Delphi. В Delphi определено около 100 стандартных типов сообщений. В соответствии с правилами Windows сообщение состоит из нескольких полей. Первое поле обычно называется Msg. Оно должно содержать индекс сообщения – 16-разрядное целое положительное число (тип Cardinal). Далее следуют поля, содержащие передаваемые значения. Последние поля обычно используются для записи результата обработки сообщения. Они могут отсутствовать.

Например, основной тип сообщений, используемых в Delphi, определяется следующим образом:

```
Type TMessage=record
       Msg:Cardinal;
       case Integer of
       0: (WParam:LongInt;  LParam:LongInt;   Result:LongInt);
       1: (WParamLo:Word; WParamHi:Word;
            LParamLo:Word;  LParamHi:Word;
           ResultLo:Word;  ResultHi:Word);
       end;
     end;
```

Номер сообщения. Номер (или индекс) сообщения используется для идентификации сообщения в системе: он определяет вид события, о котором система уведомляет приложение (нажатие клавиш, нажатие кнопок мыши и т.д.).

При создании собственных сообщений следует учитывать, что номера с 0 до $399 зарезервированы за системой. Первый свободный номер обозначен константой WM_USER = $400, относительно которой обычно и определяются номера пользовательских сообщений:

```
Const  Mes1 = WM_USER;
       Mes2 = WM_USER+1;
```

Методы обработки сообщений. Класс, объекты которого должны принимать и обрабатывать некоторые сообщения, должен включать специальные *методы обработки* этих сообщений. При разработке этих методов необходимо учитывать специальные правила, существующие в Delphi.

Метод обработки сообщения по умолчанию является *динамическим*, причем спецификаторы dinamic или override при его описании опускаются:

```
Procedure wm<имя метода>(var Message:<тип сообщения>);
                              message <номер сообщения>;
```

Имена методов обработки некоторого сообщения, переопределяемых в иерархии классов, могут не совпадать: переопределяемый метод идентифицируется по совпадающему номеру сообщения, указываемому после специальной директивы **message**. Номер сообщения обычно задается в виде символического имени, но может указываться и целой положительной константой.

Добавление метода обработки сообщений к классу выполняется так же, как и любого другого метода:

```
type <имя класса>= class <имя класса-родителя>
        public
          Procedure wm<имя метода>(var Message:<тип сообщения>);
                                  message <номер сообщения>;
      . . .
    end;
```

Если метод обработки сообщения переопределяет уже существовавший в классе-родителе, то обычно в нем программируют только специфические действия по обработке сообщения, а затем вызывают наследуемый метод для выполнения дообработки сообщения. Обращение к наследуемому методу при этом осуществляется без указания имени метода (как упоминалось выше, эти имена могут быть различны):

```
procedure <имя класса>. wm<имя метода>;
  begin
    <специальная обработка>
    inherited;
  end;
```

В том случае, если у объекта для некоторого сообщения не определен соответствующий обработчик, то, в соответствии с правилами подключения динамических методов проверяются таблицы динамических методов базовых классов. Если обработчик некоторого сообщения не определен и в базовых классах, то проводится вызов метода DefaultHandler класса TObject, который обеспечивает «обработку по умолчанию» для этих сообщений.

Генерация сообщения. Для передачи сообщений объектам Delphi могут использоваться несколько способов.

1. Для передачи сообщения *оконному элементу управления через очередь сообщений с ожиданием завершения его обработки* используется функция:

```
function SendMessage (hWnd:Integer, Mes:Cardinal;
                              WParam, LParam:LongInt):LongInt;
```

Она возвращает результат обработки сообщения. Параметр hWnd определяет номер, под которым окно – адресат сообщения – зарегистрировано

в Windows (*дескриптор* окна). Для каждого оконного элемента управления этот номер хранится в свойстве *Handle*, определенном в классе *TWinControl*.

2. Для передачи сообщения *оконному элементу управления через очередь сообщений без ожидания завершения его обработки* используется функция:

```
function PostMessage(hWnd:Integer,Mes:Cardinal;
                              WParam,Param:LongInt):LongBool;
```

Список параметров функции совпадает со списком SendMessage, но в отличие от SendMessage PostMessage ставит сообщение в очередь сообщений и возвращает управление, не ожидая завершения обработки сообщения. Функция возвращает True, если сообщение поставлено в очередь, и False – в противном случае.

3. Для передачи сообщения элементу управления *минуя очередь* используется специальный *метод* этого элемента, определенный в классе TControl:

```
procedure Perform (Mes:Cardinal; WParam, LParam:LongInt);
```

Данный метод передает сообщение элементу управления непосредственно, поэтому указывать дескриптор окна не надо, и список параметров содержит только параметры, относящиеся к самому сообщению.

Пример 5.8. Передача/прием сообщения. Разработаем приложение, одна из форм которого пересылает некоторый текст другой форме. На рис. 5.20 представлен результат работы такого приложения.

Первая форма должна вводить строку, формировать сообщение, содержащее адрес этой строки, и пересылать его второй форме. Вторая – принимать сообщение и высвечивать в специальном элементе управления переданную строку.

Прежде всего, в интерфейсной части модуля Unit1 определим тип нового сообщения и константу, определяющую его номер. Далее в классе TForm1 зарезервируем поле для хранения дескриптора второй формы и организуем пересылку сообщения в очередь по нажатию кнопки SendButton:

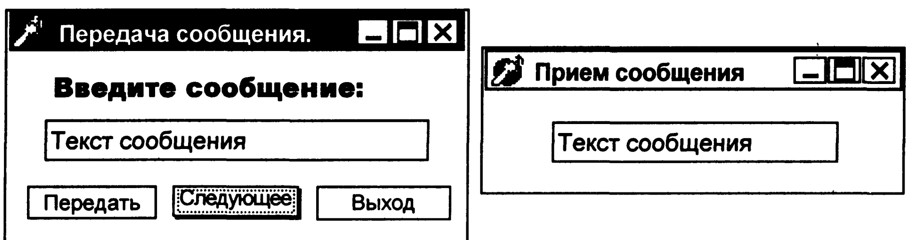


Рис. 5.20. Вид окон приложения «Передача/прием сообщений»

```
Unit Unit1;
 Interface
 Type MyMessage=Record
          Msg:Cardinal;       {номер сообщения}
          PString:PChar;      {адрес строки сообщения}
          Result:LongInt;     {поле для записи результата обработки}
        End;
  Const WM_MYMESSAGE=WM_USER; {первый номер из диапазона
                               пользовательских номеров}
  Type
   TForm1 = class(TForm)
          . . .
             public  SecondHandle:Integer;    {переменная для хранения
                                                дескриптора второй формы}
   end;
 Implementation
  Procedure TForm1. SendButtonClick(Sender: TObject);
   Begin
    SendMessage(SecondHandle,WM_MYMESSAGE,
        Longint(MessageEdit.Text),0);{генерация и пересылка сообщения}
     . . .
   End; . . .
```

Вторая форма должна принимать и обрабатывать новое сообщение. Соответственно, класс TForm2 должен включать метод обработки этого сообщения. Кроме этого, при создании формы ее дескриптор должен запоминаться в первой форме:

```
Unit Unit2;
 Interface
 Uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
       logs, StdCtrls, Unit1{содержит описание типа сообщения};
 Type  TForm2 = class(TForm)
          MessageEdit: TEdit;
         procedure FormCreate(Sender: TObject);
      public
        Procedure WMMyMessage(var Msg:MyMessage);
            MESSAGE WM_MYMESSAGE;{метод обработки сообщения}
    end;
 Var  Form2: TForm2;
Implementation
 {$R *.DFM}
 Procedure TForm2.FormCreate(Sender: TObject);
  Begin
```

```
        Form1.SecondHandle:=Handle; {запомнить дескриптор окна второй
                                                        формы}
   End;
 Procedure TForm2.WMMyMessage(var Msg:MyMessage);
   Begin
     MessageEdit.Text:=Msg.PString;         {вывести сообщение}
   End;
End.
```

Аналогично для передачи сообщения можно было бы использовать функцию PostMessage:

```
PostMessage(SecondHandle,WM_MYMESSAGE,
                              Longint(MessageEdit.Text),0);
```

Если с той же целью использовать метод Perform, то необходимо

1) в секции реализации модуля Unit1 разрешить использование модуля Unit2:

```
Uses Unit2;
```

2) для передачи сообщения использовать метод объекта Form2:

```
Form2.Perform (WM_MYMESSAGE,Longint(MessageEdit.Text),0);
```

При этом поле для хранения дескриптора в TForm1 становится ненужным.

Создание событий. При желании разработчик может, определяя свой метод обработки сообщений, предусмотреть в нем возможность вызовов обработчиков событий.

Как уже говорилось, событие реализуется как свойство процедурного типа. Соответственно, определив событие, мы должны предусмотреть его обработчик. В Delphi все события обычно имеют имена, начинающиеся с префикса «On»: OnClick, OnCreate и т.д. Для проверки наличия обработчика события используется специальная функция

```
function Assigned(var P): Boolean;
```

Эта функция проверяет, присвоено ли какое-либо значение переменной процедурного типа. Функция возвращает True, если присвоено, и False – в противном случае.

Пример 5.9. Создание события. Создадим событие в обработчике сообщения, рассмотренном в предыдущем примере. МодульUnit1 при этом не изменится.

В модуле Unit2 определим тип обработчика события TPCharEvent, получающего два параметра: стандартный параметр Sender – адрес объекта-

инициатора события и специальный параметр MyString – адрес строки, пересылаемый в сообщении. Затем объявим событие OnPChar и метод – обработчик этого события PCharProc. Подключение метода – обработчика события выполним при создании формы FormCreate.

```
Unit Unit2;
 Interface
 Uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
   logs,  StdCtrls, Unit1;
 TypeTPCharEvent=procedure(Sender: TObject;MyString:PChar) of
   object;TForm2 = class(TForm)
        MessageEdit: TEdit;
        procedure FormCreate(Sender: TObject);
     private
        FOnPChar:TPCharEvent;      {поле процедурного типа}
     public
        property OnPChar:TPCharEvent read FOnMessage
                      write FOnMessage;  {свойство-событие OnPChar }
        Procedure PCharProc(Sender: TObject; MyString:PChar);
                                          {обработчик события OnPChar}
        Procedure WMMyMessage(var Msg:MyMessage);
                                          MESSAGE WM_MYMESSAGE;
   end;
 Var  Form2: TForm2;
Implementation
 {$R *.DFM}
 Procedure TForm2.FormCreate(Sender: TObject);
  Begin
    Form1.SecondHandle:=Handle;
    OnPChar:= PCharProc;    {подключение обработчика события}
  End;
 Procedure TForm2.WMMyMessage(var Msg:MyMessage);
  Begin
    if Assigned(OnPChar) then   {если обработчик события определен, то}
              OnPChar (Self, Msg.PString);       {выполнить его}
  End;
 Procedure TForm2.PCharProc(Sender: TObject;MyString:PChar);
   Begin   MessageEdit.Text:=MyString;  End;
End.
```

Обработка сообщений компонентов VCL. Библиотека VCL Delphi использует достаточно сложные трассы передачи сообщений между компонентами и формой. Это позволяет более гибко организовать их обработку, так как сообщение может быть обработано на любом этапе.

Например, сообщение WM_KeyDoun, переданное оконному управляющему элементу класса TEdit, принадлежащему некоторой форме, обрабатывается следующим образом (рис. 5.21).

Сначала сообщение WM_KeyDoun поступает в приложение. Получив это сообщение, приложение генерирует сообщение Cn_KeyDoun для элемента управления редактированием. Получив это сообщение элемент управления редактированием генерирует сообщение Cm_KeyDoun сначала самому себе, а затем передает его элементу родителю, пока оно не будет передано форме. Если форма возвращает 0 (т.е. отказывается обрабатывать это сообщение), то элемент управления редактированием возвращает 0 сам себе, а затем 0 – приложению.

Получив 0 в качестве результата, приложение передает сообщение WM_KeyDoun элементу управления редактированием, который и обрабатывает его по умолчанию.

Управление циклом обработки сообщений. Цикл обработки сообщений приложения в Delphi скрыт в методе Run класса TApplication. Это метод получает управление после инициализации приложения и построения его окон и выполняется следующим образом. Сообщение, извлеченное из очереди методом HandleMessage, передается соответствующему оконному элементу управления для обработки. После завершения обработки сообщения из очереди извлекается следующее сообщение, и т.д., до получения сообщения о

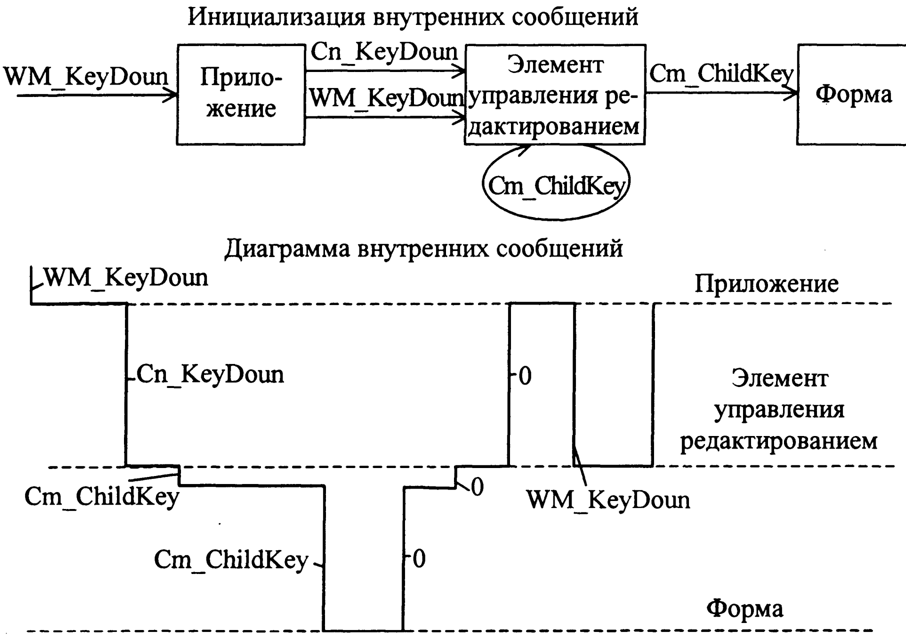


Рис. 5.21. Генерация внутренних сообщений

прекращении работы приложения WM_QUIT. При обработке сообщения WM_QUIT свойство Terminated устанавливается равным True и цикл обработки сообщений завершается:

```
... repeat HandleMessage
    until Terminated;...
```

Если сообщение инициирует длительную обработку, то процесс извлечения сообщений из очереди блокируется до ее завершения. Для пользователя это выражается в том, что приложение становится не доступным на время выполнения обработки. В таких случаях необходимо, чтобы пользователь мог удостовериться, что приложение работает, сориентироваться, какая часть работы уже выполнена, и при желании прекратить слишком длинную обработку.

Для отображения длительной обработки используют специальные элементы TProgressBar или анимацию. Изменения в окне приложения показывают пользователю, что оно выполняет какие-то действия.

Для того чтобы приложение получило возможность обрабатывать сообщение из очереди, оно должно в цикле длительной обработки периодически вызывать метод **ProcessMessages** класса TAplication. Этот метод прерывает выполнение приложения для выборки сообщений из очереди, после чего возвращает управление прерванной программе.

Для прекращения длительной обработки приходится использовать специальные приемы, так как просто инициация сообщения WM_QUIT (например, при нажатии кнопки завершения приложения) не приводит к завершению приложения до окончания обработки. Это связано с тем, что при вызове ProcessMessages не происходит возврата в цикл обработки сообщений и, следовательно, не анализируется значение свойства Terminated, устанавливаемое при нажатии кнопки завершения приложения.

Пример 5.10. Прерывание длительной обработки. Рассмотрим приложение, которое выводит в окно некоторые числа. Процесс вывода чисел отображается специальным элементом класса TProgressBar, который показывает, какая часть процесса уже завершена. При желании пользователь может остановить процесс нажатием специальной кнопки Прервать (рис. 5.22).

Рис. 5.22. Вид главного окна приложения «Прекращение длительной обработки»

Кнопку Прервать и окно индика-тора будем создавать динамически в процессе выполнения программы. Для организации прерывания обработки добавим в класс TForm1 свойство Cancel, которое будет устанавливаться равным True при нажатии кнопки Прервать. Проверка этого свойства позволит организовать досрочный выход из цикла по желанию пользователя.

```
Unit Unit1;
 Interface
  Uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
            logs,  StdCtrls, ComCtrls;
  Type
  TForm1 = class(TForm)
     StartButton: TButton;    ExitButton: TButton;
     procedure StartButtonClick(Sender: TObject);
     procedure ExitButtonClick(Sender: TObject);
     procedure ButtonClick(Sender: TObject);
   private    Cancel:Boolean;
  end;
 Var  Form1: TForm1;
Implementation
 {$R *.DFM}
 Procedure TForm1.StartButtonClick(Sender: TObject);
  Var i,j:integer;      ProgressBar: TProgressBar;      Button:TButton;
  Begin
    ProgressBar := TProgressBar.Create(Self);{создать вспомогательный
                              объект TProgressBar, объявляя основным Form1}
   Button:=TButton.Create(Self);  {создать вспомогательный объект-кнопку,
                                        объявляя основным Form1}
   ProgressBar.Parent := Self;    {объявить Form1 старшим }
   ProgressBar.Left:=30;  ProgressBar.Top:=45;{определить координаты
                                                 изображения}
   Button.Parent:=Self;              {объявить Form1старшим}
   Button.Caption:='Прервать';       {определить название кнопки}
   Button.Left:=60;     Button.Top:=65;       {определить координаты
                                                 изображения}
   Button.OnClick:=ButtonClick;      {подключить обработчик нажатия
                                              кнопки}
  for j:=0 to 9 do
    begin
    for i:=1 to 10000 do
      begin
        Canvas.TextOut(90,20,IntToStr(i));
        Application.ProcessMessages;     {прервать обработку, чтобы
                              проверить наличие сообщений в очереди}
        if Cancel then break;        {если установлено свойство Cancel, то
                                        прекратить обработку}
      end;
```

```
        ProgressBar.StepIt; {вывести в окно изображения ProgressBar очередной
                    закрашенный прямоугольник}
        if Cancel then break; {если установлено свойство Cancel, то прекратить
                                обработку}
      end;
    ProgressBar.Free;  {уничтожить объект TProgressBar}
    Button.Free;       {уничтожить кнопку TButton}
    Canvas.TextOut(70,20,'Вывод завершен');
  end;
 Procedure TForm1.ButtonClick; {обработчик нажатия на кнопку Прервать}
    Begin  Cancel:=true; End;  {устанавливаем свойство Cancel}
 Procedure TForm1.ExitButtonClick(Sender: TObject);
     Begin  Close; End;
End.
```

5.8. Обработка исключений

Во время выполнения программы могут возникнуть ситуации, в результате которых нарушается работа программы: обращение к несуществующему файлу или устройству, ошибка «деление на нуль», обращение по несуществующему адресу и т.д. Причиной таких аварийных ситуаций могут стать как некорректные действия пользователя, так и ошибки в самой программе.

Стандартные действия системы в ответ на обнаружение подобной ситуации заключаются в аварийном завершении программы с фиксацией причины в виде кода ошибки. Разработчики программного обеспечения должны максимально предусматривать возможность возникновения аварийных ситуаций, чтобы их появление не приводило к нарушению работы программы. Этой цели посвящается существенная доля всех операторов программы.

Delphi предлагает специальный механизм управления аварийными ситуациями, который позволяет «перехватить» аварийное завершение программы, определить его причину и далее, либо устранить аварийную ситуацию и продолжить работу, либо завершить работу системы с выдачей полной информации об ошибке и сохранением всех данных. Этот механизм реализуется через средства обработки *исключений.*

В основе механизма обработки исключений лежит автоматическое формирование специального блока информации при обнаружении аварийной ситуации. Доступ к этому блоку осуществляется через объект специального класса, наследуемого от базового класса всех исключений Exception.

Сравним обычный подход и подход, предполагающий использование исключений на примере обработки возможной ошибки «деление на ноль»:

Традиционный подход	Использование исключений
if n<>0 then x:=A/n else <действия по устранению ошибки >	**try** x:=A/n; **except** **on EDivByZero do** <действия по устранению ошибки > **end;**

Сравнение показывает, что в отличие от традиционного подхода, при использовании исключений действия по обработке аварийной ситуации не включаются в основной алгоритм, а группируются в своей секции специальной конструкции. Кроме того, существует гарантия перехвата аварийной ситуации, тогда как при использовании традиционного подхода такой гарантии нет. Последнее связано с тем, что ситуация «деление на нуль» фиксируется схемами контроля микропроцессора при переполнении регистра результата, что может произойти и просто при большом значении делимого и маленьком значении делителя.

Механизм обработки исключений органично вписывается в ООП, позволяя реализовывать обработку исключений в разрабатываемых классах.

К средствам обработки исключений относятся:

– специальные конструкции языка для разделения операторов основной части программы и операторов обработки исключений,

– иерархия классов различных исключений, определенная в Delphi;

– оператор генерации исключения;

– операторы обработки исключений.

Структура фрагментов с исключениями. Фрагменты программ, использующих исключения, оформляются с использованием двух специальных конструкций **try ...finally** и **try ... except**, которые определяются следующим образом:

```
try <операторы, при выполнении которых могут возникнуть исключения>
finally <операторы, которые выполняются после предыдущих или при
    возникновении исключения>
end;

try <операторы, при выполнении которых могут возникнуть исключения>
except <операторы, которые выполняются только при возникновении
    исключения>
end;
```

Использование этих конструкций позволяет выделить операторы обработки исключений в особые группы.

Первая конструкция используется, если необходимо определить группу операторов, которые должны завершить выполнение какой-либо обработки независимо от того, будет сформировано исключение или нет. Например, чтобы

закрыть открытые файлы или удалить с экрана форму.

Вторая конструкция используется, если при возникновении исключения необходимо выполнить специальные действия, которые аннулируют последствия возникновения исключительной ситуации.

Конструкции перехвата исключений могут быть вложены одна в другую в любом порядке.

Создание исключений. Базовым классом для всех исключений является класс Exception. Этот класс предусматривает достаточно большое количество конструкторов, которые формируют объекты различной структуры. Основные свойства класса Exception – свойства Сообщение и Контекст помощи. В Delphi 4 этот класс описывается следующим образом:

```
Type    Exception = class(TObject)
 private
   FMessage: string;
   FHelpContext: Integer;
 public
   Constructor Create(const Msg: string);{объект содержит сообщение,
        заданное строкой}
   Constructor CreateFmt(const Msg: string;
        const Args: array of const); { объект содержит сообщение, заданное
        строкой, дополненной форматируемыми параметрами}
   Constructor CreateRes(Ident: Integer; Dummy: Extended = 0); {объект содер-
        жит сообщение, получаемое из файла ресурса по его идентификатору}
   Constructor CreateResFmt(Ident: Integer;
        const Args: array of const); {объект содержит сообщение, получаемое из
        файла ресурса по его идентификатору, причем строка сообщения может
        дополняться форматируемыми параметрами}
   Constructor CreateHelp(const Msg: string; AHelpContext: Integer); {объект
        содержит сообщение и контекст помощи}
   Constructor CreateFmtHelp(const Msg: string;
        const Args: array of const; AHelpContext: Integer); {объект содержит
        сообщение с форматируемыми параметрами и контекст помощи}
   Constructor CreateResHelp(Ident: Integer; AHelpContext: Integer); {объект
        содержит сообщение, получаемое из файла ресурса, и контекст помощи}
   Constructor CreateResFmtHelp(Ident: Integer;
        const Args: array of const; AHelpContext: Integer); { объект содержит
        сообщение с форматируемыми параметрами, получаемое из файла
        ресурса, и контекст помощи}
  property HelpContext: Integer
        read FHelpContext write FHelpContext; {контекст помощи}
 property Message: string read FMessage write FMessage; {строка сообщения}
end;
```

Основное назначение класса исключения – идентификация групп ошибок: отнесение исключения к тому или другому классу определяет *способ обработки* данного исключения (см. далее).

Потомками класса Exception являются, например, следующие классы исключений:

EDivByZero = *class*(*EIntError*); {деление на ноль в целочисленной арифметике}
ERangeError = *class*(*EIntError*); {обращение к элементам массива по несуществующим индексам}
EIntOverflow = *class*(*EIntError*); {переполнение в целочисленной арифметике}
EMathError = *class*(*EExternal*); {ошибки арифметики с плавающей точкой}
EInvalidOp = *class*(*EMathError*); {неверный операнд)
EZeroDivide = *class*(*EMathError*); {деление на ноль в арифметике с плавающей точкой}
EOverflow = *class*(*EMathError*); {переполнение в арифметике с плавающей точкой}
EUnderflow = *class*(*EMathError*); {исчезновение порядка в арифметике с плавающей точкой}

При необходимости разработчик может определить собственные исключения, наследуя их от соответствующего класса. Желательно, чтобы имена исключений, создаваемых разработчиком, начинались с буквы «Е».

Генерация исключений. Исключения могут генерироваться автоматически (при обнаружении той или иной аварийной ситуации операционной системой) и программно (по мере надобности). Для программной генерации исключений используется специальный оператор **raise**, за которым обычно следует вызов одного из конструкторов класса-исключения, например:

if n=0 then ***raise*** *EDivByZero.Create('Количество отрезков равно 0.');*

В приведенном примере генерируется исключение класса EDivByZero. При этом конструируется специальный объект, особенностью которого является отсутствие идентификатора. Память под этот объект отводится системой автоматически.

Обработка исключений. Обработка исключений выполняется в секции **except** конструкции try ... except с помощью специальной конструкции типа case:

```
except
  on <тип исключения> do <оператор>;
  on <тип исключения> do <оператор>;
    . . .
  on <тип исключения> do <оператор>;
```

```
        else <оператор>
    end;
```

Таким образом, для каждого типа исключения может быть предусмотрена своя обработка.

Поскольку объекты исключений создаются без имени, существуют специальные способы организации доступа к полям и методам объекта-исключения.

1-й способ. Можно использовать формальную переменную, которая связывается с объектом-исключением уже в процессе обработки. Эта переменная описывается только внутри варианта on и доступна также только внутри одной альтернативы:

```
try . . .
except
on E:EDivByZero do  {объявление переменной с указанием типа}
   begin
        <переменная E доступна и может использоваться для обращения к
           полям и методам объекта, например,  E.Message>
   end;
else <переменная E – не доступна>
end;
```

2-й способ. Можно использовать функцию

function ExceptionObject:TObject;

Эта функция возвращает объектную ссылку на текущее исключение, при отсутствии исключений она возвращает nil:

```
try <контролируемые действия>
except
 on EDivByZero do
   ShowMessage(EDivByZero(ExceptionObject).Message); {используется явное
              преобразование типов}
end;
```

Если используется вложение конструкций try ... except, то можно перепоручить обработку исключения секции except внешней конструкции, указав в секции except внутренней конструкции оператор raise:

```
try . . .
  try  . . .
  except
    on <тип исключения> do <оператор>;
```

```
      . . .
    else raise;
  end
except
  <обработка остальных исключений>
end;
```

Возможны ситуации, когда необходимо знать адрес фрагмента, при выполнении которого фиксируется аварийная ситуация. С этой целью можно использовать специальную переменную

var ErrorAddr: Pointer

При генерации исключения для помещения адреса ошибки в это поле используется конструкция **raise ... at ...**, например:

raise Exception.Create(' Исключение с адресом') ***at ErrorAddr****;*

Пример 5.11. Использование исключений (класс «Динамический массив» – вариант 2). Проиллюстрируем использование исключений на примере разработки класса «Динамический массив», рассмотренном в примере 5.4.

Для того чтобы данный класс можно было безопасно использовать в приложениях, необходимо предусмотреть в нем обработку аварийных ситуаций. Такая обработка предусматривалась в примере по традиционной схеме. Используем с той же целью исключения.

Анализ алгоритмов методов данного класса позволяет определить, что при использовании объектов данного класса могут возникнуть следующие аварийные ситуации:

1) попытка записать элемент за пределами выделенной памяти;

2) попытка обращения (чтения) или записи элемента за пределами используемой части массива, за исключением добавления элемента за последним;

3) попытка преобразовать в число пустую строку или строку, содержащую недопустимые элементы.

Эти ситуации будут обнаружены в трех методах: в методе SetEl, в методе GetEl и в методе InputMas.

Соответственно, описываем в модуле новые исключения и генерируем их в вышеуказанных методах:

```
Unit MasByte;
 Interface
  Uuses SysUtils,Dialogs, Grids;
  Type
```

```
  TMas=array[1..255] of byte;
  TMasByte = class(TObject)
    private
      ptr_an: ^TMas;     {указатель на массив}
      len:byte;          {максимальная длина массива}
      Name:string; {для выдачи диагностики будем хранить имя объекта}
      Procedure SetEl(Ind:byte;m:byte);
      Function GetEl(Ind:byte):byte;
    public
      n:Byte;            {реальный размер массива}
      Constructor Create(an:byte;aName:string);
      Destructor Destroy;override;
      Property Mas[Ind: byte]:byte read GetEl write SetEl;default;
      Procedure Modify(Ind:byte; Value:byte);
      Procedure Insert(Ind:byte; Value:byte);
      Function Delete(Ind:byte):byte;
      Function InputMas(Grid:TStringGrid;I,J:integer):boolean;
      Procedure OutputMas(Grid:TStringGrid;I,J:integer);
    end;
  EInputError=class(Exception); {дополнительное исключение}
Implementation
Constructor TMasByte.Create;
  Begin
    inherited Create;
    GetMem(ptr_an,an);
    len:=an;       Name:=aName;
  End;
Destructor TMasByte.Destroy;
  Begin
    FreeMem(ptr_an);
    inherited Destroy;
  End;
Procedure TMasByte.SetEl(Ind:byte;m:byte);
 Begin
  if Ind<=len then
   if Ind<=n then ptr_an^[Ind]:=m
   else
     raise ERangeError.CreateFmt('В массиве %s нет %d-го
               элемента.',[Name,Ind])    {генерация исключения}
  else
    raise ERangeError.CreateFmt('В массиве %s можно разместить
       только %d элементов.',[Name,Len]); {генерация исключения}
End;
```

```
Function TMasByte.GetEl(Ind:byte):byte;
  Begin
   if Ind<=n then Result:=ptr_an^[Ind]
   else
     raise ERangeError.CreateFmt( 'В массиве %s нет %d-го элемента.',
                          [Name,Ind]); {генерация исключения}
  End;
Function TMasByte.InputMas(Grid:TStringGrid;I,J:integer):boolean;
 Var k:byte; x,er_code:integer;
 Begin
  with Grid do
   begin
    k:=0;
    Result:=true;
    while (Cells[k+I,J]<>'')and Result do
     begin
      Val(Cells[k+I,J],x,er_code);
      if er_code=0 then
        if x<=255 then Insert(k+1,x)
        else
          begin
           raise EInputError.Create( 'Значение не может превышать 255.');
                                  {генерация исключения}
           Result:=false;
           exit;
          end
        else
       begin
         raise EInputError.Create( 'В строке обнаружены недопустимые
                                  символы.'); {генерация исключения}
         Result:=false;
         exit;
       end;
     k:=k+1;
    end;
   OutputMas(Grid,I,J);
  end;
 End;
   . . .       {тексты остальных методов не изменились}
end.
```

Теперь при работе с объектами данного класса следует предусматривать обработку исключений, которые могут быть генерированы в процессе выполнения. Например, вызов функции ввода элементов должен выглядеть следующим образом:

```
A:=TMasByte.Create(10,'A'); {конструируем объект A}
try A.InputMas(StringGrid,0,0); {пробуем ввести массив}
except
   on E:EInputError do   {если обнаружено исключение ввода}
            MessageDlg(E.Message,mtInformation,[mbOk],0);
end;
```

Аналогично, при попытке записать что-либо в массив следует предусматривать возможность возникновения аварийной ситуации:

```
try   A[Ind]:=Value;  {пробуем занести значение в массив}
except
    on E:ERangeError do  {если обнаружено исключение «неверный
                                              индекс»)
            MessageDlg(E.Message,mtInformation, [mbOk], 0);
end;
```

Вопросы для самоконтроля

1. Дайте определение класса, принятое в Delphi Pascal. Какие новые возможности определения классов появились в этом языке программирования по сравнению с Pascal 7.0? Сравните их с аналогичными средствами, существующими в C++3.1?

2. Какие виды полиморфизма реализованы в данной среде? Дайте определение абстрактным и динамическим методам? Поясните, чем они отличаются от обычных виртуальных методов. Определите сущность перегрузки методов.

3. Определите понятие «свойство». С какой целью целесообразно использовать механизм свойств? Приведите примеры.

4. Что такое «информация о типе времени выполнения»? Зачем она используется? Дайте определение метакласса и поясните, для чего он может быть использован. Какую роль играют методы класса и почему их можно вызывать без указания имени объекта?

5. Поясните сущность понятия «делегирование методов». Какие средства должен включать язык, в котором возможна реализация делегирования?

6. Как построена библиотека VCL? Чем различаются отношения «основной/вспомогательный» и «старший/младший»? Как их можно использовать?

7. Какие средства создания сообщений предлагаются средой Delphi? В каких случаях возникает необходимость создания новых сообщений? Как описывается обработчик сообщений? Как генерировать новые события?

8. Какие ситуации попадают под понятие «исключительные»? Почему возникла необходимость создания средств обработки исключений? Поясните процесс создания/обработки исключений. Перечислите средства, позволяющие реализовать данный процесс.

6. ОБЪЕКТНАЯ МОДЕЛЬ C++ BUILDER

Объектная модель C++ Builder несколько отличается от первоначальной объектной модели C++, описанной в главе 3. Прежде всего, она базируется на современной усложненной модели, используемой в последних версиях языка C++, т.е. поддерживает пространства имен, исключения и специальные средства преобразования типов.

Кроме этого, C++ Builder использует библиотеку классов VCL, разработанную для среды Delphi и основывающуюся на объектной модели Delphi Pascal. Следовательно, при создании C++ Builder необходимо было согласовать конструкции C++ и Pascal Delphi, а также механизмы реализации этих конструкций, обращая особое внимание на те возможности, которые есть в Delphi Pascal, но отсутствуют в C++. В результате в объектную модель C++ были добавлены:

– возможность создания специальных секций для описания опубликованных элементов класса и элементов, реализующих OLE-механизм;

– средства объявления свойств;

– определения специальных классов, моделирующих стандартные типы данных Delphi Pascal (множества, строки и т.д.);

– возможность определения указателей на методы (**_ _closure**);

– специальный модификатор (**_ _declspec**), посредством которого реализуются, например, динамические методы Delphi Pascal.

Различие объектных моделей и их реализаций в C++ и Delphi Pascal не позволяет обеспечить полную совместимость классов, разработанных в этих языках. Поэтому разработчики C++Builder обеспечили возможность создания двух типов классов: обычные классы C++ с расширенными возможностями и VCL-совместимые классы – для работы с библиотекой визуальных компонент VCL.

6.1. Расширение базовой объектной модели C++

Как уже говорилось выше, объектная модель C++ Builder включает ряд новых (по сравнению с Borland C++ 3.1) средств. Это средства:

– определения пространств имен;

– описания указателей на методы;

– определения и переопределения типа объекта;

– описания свойств.

Пространство имен. Большинство сколько-нибудь сложных приложений состоит более чем из одного исходного файла. При этом возникает вероятность

дублирования имен, что препятствует сборке программы из частей. Для снятия проблемы дублирования имен в С++ был введен механизм логического разделения области глобальных имен программы, который получил название *пространства имен.*

Пространство имен описывается следующим образом:

namespase [<имя>] { <объявления и определения> }

Имя пространства имен должно быть уникальным, но может быть и опущено.

Примечание. Если имя пространства опущено, то считается, что определено неименованное пространство имен, локальное внутри единицы трансляции. Для доступа к его ресурсам используется внутреннее имя $$$.

Имена, определенные в пространстве имен, становятся локальными внутри него и могут использоваться независимо от имен, определенных в других пространствах. Таким образом, снимается требование уникальности имен программы.

Например:

```
namespace ALPHA {               // ALPHA – имя пространства имен
    long double LD;             // объявление переменной
    float f(float y) { return y; }  // описание функции
    }
```

Имя пространства имен должно быть известно во всех модулях программы, которые используют его элементы.

Пространство имен определяет область видимости, следовательно, функции, определенные в одном пространстве имен, могут без ограничений вызывать друг друга и использовать другие ресурсы, объявленные там же (переменные, типы и т.д.).

Доступ к элементам других пространств имен может осуществляться тремя способами:

– с использованием имени области в качестве квалификатора доступа, например:

```
ALPHA:: LD
ALPHA::f()
```

– с использованием *объявления using*, которое указывает, что некоторое имя доступно в другом пространстве имен:

```
namespace BETA {        ...
    using ALPHA::LD;  /* имя ALPHA::LD доступно в BETA*/ }
```

– с использованием *директивы using*, которая объявляет все имена одного пространства имен доступными в другом пространстве:

```
namespace BETA {      ...
    using ALPHA;   /* все имена ALPHA доступны в BETA*/  }
```

Каждое объявление класса в С++ образует пространство имен, куда входят все общедоступные компоненты класса. Для доступа к ним принято использовать квалификаторы доступа <имя класса>::.

Директиву using внутри класса использовать не разрешается. Применение объявления using допустимо и может оказаться весьма полезным.

Пример 6.1. Переопределение метода потомка перегруженным методом базового класса (с использованием объявления using). Описание базового и производного классов в соответствии с правилами модульного программирования в С++ выполним в файле-заголовке Object.h:

```
#ifndef ObjectH
#define ObjectH
class A
 {  public:    void func(char ch,TEdit *Edit);
 };
class B : public A
 {  public:
       void func(char *str,TEdit *Edit);
       using A::func;          // перегрузить B::func
 };
#endif
```

Реализацию методов классов поместим в файле Object.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include " Object.h"
void A::func(char ch,TEdit *Edit)                // метод базового класса
      { Edit->Text=AnsiString("символ"); }
void B::func(char *str,TEdit *Edit)              // метод производного класса
      { Edit->Text=AnsiString("строка"); }
#pragma package(smart_init)
```

Вызов нужного метода, как это принято для перегруженных функций, определяется типом фактического параметра:

```
B b;
b.func('c',Edit);  // вызов A::func(), так как параметр – символ
b.func("c",Edit);  // вызов B::func(), так как параметр – строка
```

Указатель на метод. Делегирование. В стандартном С++ существует возможность объявления указателей на функции. Аналогично можно объявлять указатели на методы, как компонентные функции определенного класса. Такое объявление должно содержать квалификатор доступа вида <имя класса>::. Вызов метода по адресу осуществляется с указанием объекта, для которого вызывается метод.

Например, если описан класс base:

```
class base
  { public:   void func(int x,TEdit *Edit);
  };
```

то можно определить указатель на метод этого класса и обратиться к этому методу, используя указатель:

```
base A;
  void (base::*bptr)(int,TEdit *); // указатель на метод класса
  bptr = & base::func;             // инициализация указателя
  (A.*bptr)(1,ResultEdit);         // вызов метода через указатель
```

Причем существенно, что указатель определен именно для методов данного класса и не может быть инициализирован адресом метода даже производного класса (не говоря уж о методах других классов):

```
class derived: public base
  { public:   void new_func(int i,TEdit *Edit);
  };
...
bptr =&derived::new_func; // ошибка при компиляции !!!
```

С помощью описателя **__closure** в С++ Builder объявляется специальный тип указателя – указатель на метод. Такой указатель помимо собственно адреса функции содержит адрес объекта, для которого вызывается метод. В отличие от обычных указателей на функции, указатель на метод не приписан к определенному классу и потому может быть инициализирован адресами методов различных классов.

Объявление указателя на метод выполняется следующим образом:

<тип> (**__closure** * <идентификатор>) (<список параметров>);

Например, для классов, описанных выше, можно выполнить следующие объявления:

```
base A;        derived B;
void (__closure *bptr)(int,TEdit *);  // указатель на метод
bptr = &A.func;          /* инициализация указателя адресом метода
                            базового класса и адресом объекта A */
bptr(1,ResultEdit);      // вызов метода по указателю
bptr = &B.new_func;  /* инициализация указателя адресом метода
                            производного класса и адресом объекта B*/
bptr(1,ResultEdit);      // вызов метода по указателю
```

Указатели на методы поддерживают сложный полиморфизм. Так, если базовый и производный классы содержат виртуальные методы, то при вызове метода по указателю определение класса объекта и, соответственно, аспекта полиморфного метода будет происходить во время выполнения программы.

Например:

```
class base
 { public:  virtual void func_poly(TEdit *Edit);
 };
class derived: public base
 { public:  virtual void func_poly(TEdit *Edit);
 };
...
base *pB;
pB=new derived;
 void (__closure *bptr)(TEdit *); // указатель на метод
bptr = &pB->func_poly;  /* инициализация указателя адресом полиморфного
                     метода и адресом объекта производного класса, нужный
                     аспект определяется во время выполнения программы */
bptr(ResultEdit);         // вызов метода по указателю
```

Указатели на метод используются для подключения обработчиков событий, но могут использоваться и для выполнения делегирования методов.

Пример 6.2. Делегирование методов (графический редактор «Окружности и квадраты»). Делегирование методов проиллюстрируем на примере разработки графического редактора «Окружности и квадраты», рассмотренного в разделе 5.5 (пример 5.6). Сначала опишем класс TFigure в файле Object.h:

```
#ifndef FigureH
#define FigureH
typedef void (__closure *type_pMetod)(TImage *);
class TFigure
{private: int x,y,r;
```

```
  type_pMetod fDraw;
  public:
    __property type_pMetod Draw ={read=fDraw,write=fDraw};
    TFigure(int, int, int, TImage *, TRadioGroup * );
    void DrawCircle(TImage *);
    void DrawSquare(TImage *);
    void Clear(TImage *);
 };
#endif
```

Описание методов класса поместим в файл Object.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "Figure.h"
TFigure::TFigure(int X, int Y, int R,
                    TImage *Image, TRadioGroup *RadioGroup)
 { x=X;   y=Y;   r=R;
   switch (RadioGroup->ItemIndex)  // определить метод рисования
     {case 0:  Draw=DrawCircle;  break;
      case 1:  Draw=DrawSquare; }
   Draw(Image);                    // нарисовать фигуру
 }
void TFigure::DrawCircle(TImage *Image)
 {Image->Canvas->Ellipse(x-r, y-r, x+r, y+r);  }
void TFigure::DrawSquare(TImage *Image)
 {Image->Canvas->Rectangle(x-r, y-r, x+r, y+r); }
void TFigure::Clear(TImage *Image)
 { Image->Canvas->Pen->Color=clWhite;
   Draw(Image);  // вызов метода по адресу, указанному в свойстве
   Image->Canvas->Pen->Color=clBlack;  }
#pragma package(smart_init)
```

Объекты класса Figure будем создавать при нажатии клавиши мыши:

```
void __fastcall TMainForm::ImageMouseDown(TObject *Sender,
                    TMouseButton Button, TShiftState Shift, int X, int Y)
 { if (Figure!=NULL) delete Figure; // если объект создан, то уничтожить
   Figure=new TFigure(X,Y,10,Image,RadioGroup);  // создать объект
 }
```

При переключении типа фигуры будем стирать уже нарисованную фигуру и рисовать фигуру другого типа:

```
void __fastcall TMainForm::RadioGroupClick(TObject *Sender)
 { if (Figure!=NULL)                    // если фигура нарисована, то
   { Figure->Clear(Image);              // стереть ее
    switch (RadioGroup->ItemIndex)      // делегировать метод
     { case 0: Figure->Draw=Figure->DrawCircle;  break;
       case 1: Figure->Draw=Figure->DrawSquare;  }
    Figure->Draw(Image);   }    // нарисовать фигуру другого типа
 }
```

Операторы определения и переопределения типа объекта. Эти операторы были включены в C++, чтобы обезопасить операцию переопределения (приведения) типов, которая программировалась следующим образом:

(<имя типа>)<имя переменной>

В теории ООП различают нисходящее и восходящее приведения типов для объектов иерархии классов. Приведение типов называют *нисходящим*, если в его результате указатель или ссылка на базовый класс преобразуется в указатель или ссылку на производный, и *восходящим*, если указатель или ссылка на производный класс преобразуется в указатель или ссылку на базовый класс.

Восходящее приведение типов никакой сложности не представляет и возможно во всех случаях. Оно используется сравнительно редко, практически только в тех случаях, когда необходимо объекту производного класса обеспечить доступ к переопределенным методам базового класса, например:

```
class A {  public:   void func(char ch); };
 class B : public A
        {  public:   void func(char *str);  };
    ...
   B b;
   b.func("c");     // вызвать B::func()
  (A)b.func('c');  // вызвать A::func(); (A)b – восходящее приведение типа
```

При выполнении нисходящего приведения типов необходима проверка, так как никакой гарантии, что указатель ссылается на адрес объекта именно данного производного класса, у нас нет. Используется же это преобразование при работе с полиморфными объектами постоянно, в связи с тем, что это единственный способ обеспечить видимость полей производного класса при работе с объектом через указатель на базовый класс (раздел 1.6).

В последних версиях C++ приведение типов выполняется с использованием специальных операторов.

Рассмотрим эти операторы.

Динамическое приведение типа: **dinamic_cast <T>(t)**.

Операнды: T – указатель или ссылка на класс или void*, t – выражения типа указателя, причем оба операнда либо указатели, либо ссылки.

Приведение типа осуществляется во время выполнения программы. Предусмотрена проверка возможности преобразования, использующая RTTI (информацию о типе времени выполнения), которая строится в С++ *только для полиморфных объектов*.

Применяется для нисходящего приведения типов полиморфных объектов, например:

```
class A {virtual ~A(){}};/*класс обязательно должен включать виртуальный
                метод, так как для выполнения приведения требуется RTTI*/
class B: public A {virtual ~B(){}};
void func(A& a)  /* функция, работающая с полиморфным объектом*/
    { B& b=dinamic_cast<B&>(a); // нисходящее приведение типов
    }
void somefunc()
    { B b;
      func(b);       // вызов функции с полиморфным объектом
    }
```

Если вызов dynamic_cast осуществляется в условной конструкции, то ошибка преобразования, обнаруженная на этапе выполнения программы, приводит к установке значения указателя равным NULL (0), в результате чего активизируется ветвь «иначе». Например:

```
if (Derived* q=dinamic_cast<Derived* p>)
            {<если преобразование успешно, то ...>}
else {<если преобразование не успешно, то ...>}
```

В данном случае осуществляется преобразование указателя на базовый класс в указатель на производный класс с проверкой правильности на этапе выполнения программы (с использованием RTTI). Если преобразование невозможно, то оператор возвращает NULL и устанавливается q=NULL, в результате чего управление передается на ветвь else.

Если вызов осуществляется в операторе присваивания, то при неудаче генерируется исключение bad_cast. Например:

```
Derived* q=dinamic_cast<Derived* p>;
```

Статическое приведение типа: **static_cast<T>(t)**.

Операнды: Т – указатель, ссылка, арифметический тип или перечисление; t – аргумент типа, соответствующего Т. Оба операнда должны быть определены на этапе компиляции. Операция выполняется на этапе компиляции без проверки правильности преобразования.

Статически можно преобразовывать:

1) целое число в целое другого типа или в вещественное и обратно:

*int i; float f=**static_cast**<float>(i);* /* осуществляет преобразование без проверки на этапе компиляции программы */

2) указатели различных типов, например:

*int *q=**static_cast**<int>(malloc(100));* /* осуществляет преобразование без проверки на этапе компиляции программы */

3) указатели и ссылки на объекты иерархии в указатели и ссылки на другие объекты той же иерархии, если выполняемое приведение однозначно. Например, восходящее приведение типов или нисходящее приведения типов не полиморфных объектов, иерархии классов которых не используют виртуального наследования:

```
class A {...}; // класс не включает виртуальных функций
class B: public A {}; // не используется виртуальное наследование
  void somefunc()
   { A  a; B  b;
     B& ab=static_cast<B&>(a);  // нисходящее приведение
     A& ba=static_cast<A&>(b);  // восходящее приведение
   }
```

П р и м е ч а н и е. Кроме описанных выше были добавлены еще два оператора приведения, которые напрямую с объектами обычно не используются. Это оператор **const_cast<T>(t)** – для отмены действия модификаторов const или volutile и оператор **reinterpret<T>(t)** – для преобразований, ответственность за которые полностью лежит на программисте.

Используя в каждом случае свою операцию преобразования типов, мы получаем возможность лучше контролировать результат.

Свойства. Механизм свойств был заимствован C++ Buider из Delphi Pascal и распространен на все создаваемые классы. В Delphi Pascal свойства использовались для определения интерфейса к отдельным полям классов (см. раздел 5.3). Синтаксис и семантика свойств C++ Builder полностью аналогичны синтаксису и семантике свойств Delphi Pascal.

Так же как в Delphi Pascal различают: простые свойства, свойства-массивы и индексируемые свойства.

П р о с т ы е   с в о й с т в а определяются следующим образом:

__property <тип свойства> <имя> = {<список спецификаций>};

Список спецификаций может включать следующие значения, перечисляемые через запятую:

read = <переменная или имя функции> – определяет имя поля, откуда читается значение свойства, или функции (метода чтения), которая возвращает это значение, если данный атрибут опущен, то свойство не доступно для чтения из программы;

write = <константа или имя функции> – определяет имя поля, куда записывается значение свойства, или процедуры (метода записи), используемой для записи значения в поле; если данный атрибут опущен, то свойство не доступно для изменения из программы;

stored = <константа или имя функции логического типа> – определяет, должно ли сохраняться значение свойства в файле формы, этот атрибут используется для визуальных и невизуальных компонент;

default = <константа> или nodefault – определяет значение по умолчанию или его отсутствие.

Пример 6.3. Простые свойства (класс Целое число). Пусть требуется разработать класс для хранения целого числа. Этот класс должен обеспечивать возможность чтения значения и его записи. Опишем доступ к полю, используемому для хранения значения, используя свойство Number целого типа. Поместим это описание в файле Object.h:

```
class TNumber
 {private: int fNum;
        int GetNum(); // метод чтения свойства
        void SetNum(aNum); // метод записи свойства
  public:
    TNumber(int aNum); // конструктор
    __property int Number={read=GetNum,write=SetNum}; // свойство
 };
```

Соответственно реализацию методов этого класса поместим в файл Object.cpp:

```
#include "Object.h"
#pragma package(smart_init)
TNumber::TNumber(int aNum) {  SetNum(aNum); }
int TNumber::GetNum() {  return fNum; }
void TNumber::SetNum(int aNum) {  fNum=aNum; }
```

Для тестирования методов класса выполним следующие действия:

```
TNumber * pNumber;          // объявить переменную-указатель
pNumber=new TNumber(8);     // создать динамический объект
int  i=pNumber->Number;     // читать значение
pNumber->Number=6;          // изменить значение
delete pNumber;             // уничтожить объект
```

При желании мы могли бы запретить изменение значения свойства из программы, описав свойство следующим образом:

__property int Number={read=GetNum}; // свойство "только для чтения"

Тогда при компиляции строки

pNumber->Number=6;

получили бы сообщение об ошибке.

С в о й с т в а – м а с с и в ы объявляются с указанием индексов:

__property <тип> <имя> [<тип и имя индекса>] = {<список атрибутов>};

Индексов может быть несколько. Каждый индекс записывается в своих квадратных скобках. В качестве индекса может использоваться любой скалярный тип С++.

В списке атрибутов свойства-массива атрибуты stored и default не используют, в нем указывают только методы чтения и записи, например:

__property int Mas[int i][int j]={read=GetMas, write=SetMas}; {объявлено свойство-массив Mas с двумя индексами}

Методы чтения и записи, указанные при описании свойства-массива, должны в списке параметров содержать столько же индексов, что и описанное свойство (см. раздел 5.3). Фактические значения индексов, указанные при обращении к свойству, будут переданы в качестве фактических параметров методам чтения или записи свойства. Они могут использоваться произвольным образом для связи программы с соответствующими полями объекта.

Пример 6.4. Свойства-массивы (класс Динамический массив). В данном примере реализуется на С++ определение класса Динамический массив, рассмотренного в разделе 5.8. При разработке методов класса использованы исключения, средства создания и обработки которых в С++ Builder рассмотрены в разделе 6.2.

Объявление класса – помещается в файл заголовка array.h:

```
class TMasByte
{ private:
   unsigned char* ptr_an;          // указатель на массив
   unsigned char len;              // максимальная длина массива
   void SetEl(short Ind, unsigned char m); // метод записи
   unsigned char GetEl(short Ind);         // метод чтения
 public:
   unsigned char n;                // реальная длина массива
```

```
   TMasByte(unsigned char alen); // конструктор
   ~TMasByte();                  // деструктор
  __property unsigned char Mas[short Ind]={read=GetEl, write=SetEl};
   void Modify(short Ind,unsigned char Value); // изменить значение
   void Insert(short Ind,unsigned char Value);  // вставить значение
   unsigned char Delete(short Ind);             // удалить значение
   void InputMas(TStringGrid* Grid,int i,int j); // ввести из таблицы
   void OutputMas(TStringGrid* Grid,int i,int j); // вывести в таблицу
};
```

Реализация методов помещается в файл array.cpp:

```
TMasByte::TMasByte(unsigned char alen)
   { ptr_an=new unsigned char[alen]; len=alen; n=0;}
TMasByte::~TMasByte()
  { delete [] ptr_an;}
void TMasByte::SetEl(short Ind, unsigned char m) // метод записи
{ if (Ind<len)
    if (Ind<n)  ptr_an[Ind]=m;
    else throw ("Запись за пределами реального размера массива.");
 else throw ("Запись за пределами отведенного пространства."); }
unsigned char TMasByte::GetEl(short Ind)        // метод чтения
 {  if (Ind<n) return ptr_an[Ind];
    else throw ("Чтение за пределами реального массива.") ; }
void TMasByte::Modify(short Ind,unsigned char Value)
 {  Mas[Ind]=Value;}
void TMasByte::Insert(short Ind,unsigned char Value)
{ n++;
 for (short i=n-1;i>Ind;i—) Mas[i]=Mas[i-1];
 Mas[Ind]=Value;}
unsigned char TMasByte::Delete(short Ind)
 { unsigned char Result=Mas[Ind];
  for (short i=Ind;i<n-1;i++) Mas[i]=Mas[i+1]; n—;
  return Result; }
void TMasByte::InputMas(TStringGrid* Grid,int i,int j)
{ int k=0;
 while (Grid->Cells[k+i][j].Length())
  { try { unsigned char x=StrToInt(Grid->Cells[k+i][j]);
         if (x<255) Insert(k,x);
         else throw ("Значение не может превышать 255");
         k++;   }
  catch (EConvertError&)
     {throw ("В строке обнаружены недопустимые символы");}
  }
```

```
  OutputMas(Grid,i,j);}
void TMasByte::OutputMas(TStringGrid* Grid,int i,int j)
{  if (n+i>Grid->ColCount) Grid->ColCount=n+1;
   for (int k=0;k<Grid->ColCount;k++)
       if (k<n)Grid->Cells[i+k][j]=IntToStr(Mas[k]);
       else Grid->Cells[i+k][j]="";  }
```

Обращение к методам класса TMasByte в тестирующей программе выполняется следующим образом:

```
TMasByte* A;                       // объявить переменную-указатель
A=new TMasByte(10);                // конструировать массив
A->InputMas(DataStringGrid,0,0);   // ввести элементы из таблицы
A->Insert(Ind,Value);              // вставить элемент
A->OutputMas(DataStringGrid,0,0);  // вывести элементы в таблицу
delete A;                          // уничтожить объект
```

Индексируемые свойства описываются с дополнительным атрибутом index, за которым следует константа или выражение целого типа:

__property <тип> <имя> = {<список атрибутов>, **index** = <константа>};

Индекс в списке атрибутов используется в качестве дополнительного параметра методов чтения и записи, что позволяет использовать одни и те же методы для чтения и записи группы свойств, не образующих свойство-массив.

Например:

```
  private:
    int fRegion[3];
    int GetRegion(int index);
    void SetRegion(int index,int value);
  public:
    __property int Region1={read=GetRegion, write=SetRegion, index=0};
    __property int Region2={read=GetRegion, write=SetRegion, index=1};
    __property int Region3={read=GetRegion, write=SetRegion, index=2};
```

Объявлены свойства, обеспечивающие доступ по именам (псевдонимам) к элементам массива fRegion, методы чтения и записи в этом случае должны использовать указанный индекс для доступа к нужному элементу, например:

```
  int   <имя класса>::GetRegion(int index){return fRegion[index];}
  void <имя класса>::SetRegion(int index,int value){fRegion[index]=value;}
```

Свойства отличаются от обычных полей данных тем, что

– связывают с именем свойства методы чтения и записи значений;

– устанавливают для свойств значения, принимаемые по умолчанию;
– простые свойства могут храниться в файлах форм;
– могут изменять правила доступа, описанные в базовом классе.

6.2. Исключения

Как уже упоминалось в разделах 1.7 и 5.8, достаточно большая часть любой программы приходится на перехват и обработку ситуаций, при возникновении которых по каким-либо причинам нормальный процесс обработки нарушается (ввод некорректной информации, попытка читать из несуществующего файла, обнаружение ситуации «деление на нуль» и т.п.).

Использование для проектирования программы технологии ООП приводит к тому, что обычно возникновение некорректной ситуации фиксируется в одном месте (объекте) программы, а принятие решения по ней возможно лишь в другом. Для обработки таких ситуаций используют механизм исключений.

К сожалению, традиционно в С и С++ используются разные стандарты обработки исключений.

Механизм исключений С++. В языке С++ используются специальные операторы **throw**, **try** и **catch**. Первый – для генерации исключения, а два других – для организации его перехвата.

Генерация исключений выполняется в том месте программы, где обнаруживается исключительная ситуация.

Оператор **throw** имеет следующий синтаксис:

throw [<тип>](<аргументы>);

где <тип> – тип (чаще класс) генерируемого значения; если тип не указан, то компилятор определяет его исходя из типа аргумента (обычно это один из встроенных типов); <аргументы> – одно или несколько выражений, значения которых будут использованы для инициализации генерируемого объекта.

Например:

1) *throw ("неверный параметр");* /* генерирует исключение типа const char * с указанным в кавычках значением */

2) *throw (221);* /* генерирует исключение типа const int с указанным значением */

```
3)      class E {   //класс исключения
     public: int num;              // номер исключения
             E(int n): num(n){}    // конструктор класса
        }
...
throw E(5); // генерирует исключение в виде объекта класса E
```

Перехват и обработка исключений осуществляются с помощью конструкции **try** … **catch** …(**catch**…):

try {<защищенный код>}
catch (<ссылка на тип>){<обработка исключений>}

Блок операторов try содержит операторы, при выполнении которых могут возникнуть исключительные ситуации. Блоков catch может быть несколько. Каждый блок catch включает операторы, которые должны быть выполнены, если при выполнении операторов блока try были зафиксировано исключение типа, совместимого с указанным в catch. При этом:

– исключение типа T будет перехватываться обработчиками типов T, const T, T& или const T&;

– обработчики типа общедоступного базового класса перехватывают исключения типа производных классов;

– обработчики типа void* перехватывают все исключения типа указателя.

Блок catch, для которого в качестве типа указано «…», обрабатывает исключения всех типов.

Например:

```
try {<операторы>} // выполняемый фрагмент программы
catch (EConvert& A){<операторы>} /* перехватывает исключения указанного
                                    типа EConvert */
catch (char* Mes){<операторы>}   /*перехватывает исключения типа char* */
catch(...) {<операторы>}         //перехватывает остальные исключения
```

Таким образом, обработчик может перехватывать исключения нескольких типов. При этом существенным оказывается порядок объявления обработчиков исключений.

Так, обработчик типа void* не должен указываться перед обработчиком типа char*, потому что при таком порядке все исключения типа char* будут обрабатываться обработчиком void*. То же самое касается недопустимости указания обработчика исключений базового класса перед обработчиками производных классов.

Например:

```
class E{};
class EA:public E{};  ...
try {...}
catch (E& e) {...} // этот обработчик перехватит все исключения
catch (EA& e){...}  // этот обработчик никогда не будет вызван
```

Иногда оказывается, что перехваченное исключение не может быть обработано в данном месте программы. В этом случае в обработчик включается

оператор throw без параметра, в результате исключение генерируется повторно с тем же объектом, что и первый раз. При обработке этого исключения возобновляется просмотр стека вызовов в целях обнаружения другого обработчика данного или совместимого типов.

Например:

```
class E{};                      // класс исключения
void somefunc()
  { if(<условие> throw Out(); }// функция, генерирующая исключение
void func()
 { try      { somefunc(true); }
     catch(E& e){ if (<условие>) throw; } /* если здесь исключение
                                   обработать нельзя, то возобновляем его */ }
void mainfunc()
 { try      {func(); }
   catch(E& e){ ... } } // здесь обрабатываем исключение
```

Стек вызовов для данного примера показан на рис. 6.1.

Функция somefunc генерирует исключение. Для его обработки осуществляется обратный просмотр стека вызовов, т.е. по очереди проверяются все функции, выполнение которых не завершено. При этом обнаруживается, что вызов sumefunc() осуществлен в защищенном блоке функции func(), и, следовательно, проверяется соответствие типа исключения типу имеющегося обработчика. Тип соответствует, следовательно исключение перехвачено, но если оно не может быть обработано в данном обработчике, то исключение генерируется вновь. Теперь в поисках обработчика исключения проверяется следующая незавершенная функция – mainfunc(). В этой функции обнаруживается, что вызов func () выполнялся в защищенном блоке. При проверке связанного с ним блока catch выясняется, что данное исключение перехватывается и обрабатывается.

Использование имени переменной в качестве параметра оператора catch позволяет операторам обработки получить доступ к аргументам исключения через указанное имя. Например:

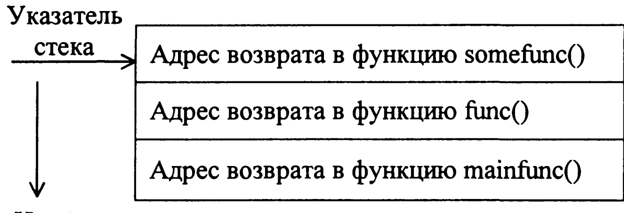


Рис. 6.1. Содержимое стека вызовов при возникновении исключений

```
class E   //класс исключения
{ public: int num;              // номер исключения
      E(int n): num(n){}    // конструктор
}
...
throw E(5);          // генерируемое исключение
...
catch (E& e){if (e.num==5) {...}} // получен доступ к полю
```

Полностью последовательность обработки исключения выглядит следующим образом:

1) при генерации исключения происходит конструирование временного объекта исключения;

2) выполняется поиск обработчика исключения;

3) при нахождении обработчика создается копия объекта с указанным именем;

4) уничтожается временный объект исключения;

5) выполняется обработка исключения;

6) уничтожается копия исключения с указанным именем.

Поскольку обработчику передается копия объекта исключения, желательно в классе исключения со сложной структурой предусмотреть копирующий конструктор и деструктор.

Иногда бывает удобно указать при объявлении функции, какие исключения она может генерировать. Именно для этих исключений программист будет предусматривать обработчики при вызове функции. Указание генерируемых исключений осуществляется в *спецификации исключений*:

throw(<тип>,<тип>...).

Например:

```
void func () throw(char*,int){...} /*данная функция может генерировать
                                      исключения типов char* и int   */
```

При организации перекрытия виртуальных методов следует учитывать, что спецификация исключений не считается частью типа функции и, следовательно, ее можно изменить:

```
class ALPHA
{public:
 struct ALPHA_ERR {}; virtual void vfunc() throw (ALPHA_ERR) {}
                                 // спецификация исключения
};
```

```
class BETA : public ALPHA
{ public:
    void vfunc() throw(char *) {}   // изменение спецификации
};
```

Если в процессе выполнения программы будет сгенерировано исключение не предусмотренного спецификацией типа, то управление будет передано специальному обработчику непредусмотренных исключений. Для определения этой функции в программе используется функция **set_unexpected**:

```
void my_unexpected(){<обработка исключений> }
...
set_unexpected(my_unexpected);
```

Функция set_unexpected() возвращает старый адрес функции – обработчика непредусмотренных исключений, что позволяет организовать достаточно сложную обработку.

Если нужный обработчик при обратном просмотре стека вызовов не найден, а обработчик непредусмотренных исключений отсутствует, то вызывается функция **terminate**(). По умолчанию эта функция вызывает функцию **abort**(), которая аварийно завершает текущий процесс.

Можно установить собственную функцию завершения, используя функцию **set_terminate**():

```
void my_terminate(){<обработка завершения>}
...
set_terminate(my_terminate);
```

Функция set_terminate() также возвращает адрес предыдущей программы обработки завершения.

В качестве примера использования исключений C++ можно вернуться к тексту программы примера 6.2. Наиболее интересным является код метода ввода элементов массива из таблицы:

```
void TMasByte::InputMas(TStringGrid* Grid,int i,int j)
{ int  k=0;
  while (Grid->Cells[k+i][j].Length())
   { try { unsigned char x=StrToInt(Grid->Cells[k+i][j]);
        if (x<255) Insert(k,x);
else throw ("Значение не может превышать 255"); /* генерация исключения
                                                              – строки */
        k++;    }
    catch (EConvertError&) // исключение преобразования строки в число
      {throw ("В строке обнаружены недопустимые символы.");} /* генерация
                                                 исключения – строки*/
   }
```

В этом фрагменте генерируются исключения типа строка, которые могут быть обработаны при вызове функции, например:

```
A=new TMasByte(10);
try {A->InputMas(DataStringGrid,0,0); }
 catch (char* Mes)
       { TMsgDlgButtons Set2;
        Set2<<mbOK;
        MessageDlg(Mes,mtInformation,Set2,0);  }
```

Помимо обычной обработки исключения C++ Builder позволяет осуществлять их завершающую обработку. Операторы завершающей обработки выполняются независимо от возникновения или отсутствия исключений в защищенном коде.

Для организации завершающей обработки используется служебное слово **__finally**:

try {<защищенный код>}
__finally{<завершающая обработка>}

Механизм завершающей обработки описан далее, так как первоначально он появился в структурном управлении исключениями C.

Механизм исключений C. В языке C используется так называемое *структурное управление исключениями.*

В основе структурного управления исключениями лежат конструкции **__try...__except** и **__try...__finally**. Первая обеспечивает обычную обработку исключения, вторая – завершающую.

Обычная обработка программируется следующим образом:

__try {<защищенный код>}
__except(<фильтрующее выражение>)
   {<обработка исключений>}

Фильтрующее выражение может принимать следующие значения:

– EXCEPTION_EXECUTE_HANDLER – управление должно быть передано на следующий за ним обработчик исключения (при этом по умолчанию при обратном просмотре стека вызовов активизируются деструкторы всех локальных объектов, созданных между местом генерации исключения и найденным обработчиком);

– EXCEPTION_CONTINUE_SEARCH – проводится поиск другого обработчика;

– EXCEPTION_CONTINUE_EXECUTION – управление возвращается в то место, где было обнаружено исключение без обработки исключения (отмена исключения).

Как правило, в качестве фильтрующего выражения используется функция, которая возвращает одно из указанных выше трех значений.

Библиотека excpt.h включает также функции, позволяющие получить некоторую информацию об исключении:

GetExceptionCode() – возвращает код исключения.

GetExceptionInformation() – возвращает указатель на структуру, содержащую описание исключения.

Существует ограничение на вызов этих функций: они могут вызываться только непосредственно из блока __except(), например:

```
#include <excpt.h>
int filter_func(EXCEPTION_POINTERS *);
...
EXCEPTION_POINTERS *xp = 0;
  try {      foo();   }
        __except (filter_func(xp = GetExceptionInformation())) {/* получение
                                              информации об исключении */   }
```

или с использованием составного оператора:

```
__except((xp = GetExceptionInformation()), filter_func(xp))
```

Фильтрующая функция не может вызывать функцию GetExceptionInformation (), но результат этой функции можно передать в качестве параметра. Функция GetExceptionInformation() возвращает указатель на структуру EXCEPTION_POINTERS:

```
struct EXCEPTION_POINTERS {
    EXCEPTION_RECORD *ExceptionRecord;
    CONTEXT *Context;
};
```

Структура EXCEPTION_RECORD в свою очередь определяется следующим образом:

```
struct EXCEPTION_RECORD
{   DWORD ExceptionCode;    // код завершения
    DWORD ExceptionFlags;   // флаг возобновления
    struct EXCEPTION_RECORD *ExceptionRecord;
    void *ExceptionAddress;    // адрес исключения
    DWORD NumberParameters;  // количество аргументов
    DWORD ExceptionInformation
    [EXCEPTION_MAXIMUM_PARAMETERS];
        /* адрес массива параметров */
 };
```

Обычно фильтрующая функция обращается к информации ExceptionRecord, чтобы определить, следует ли обрабатывать исключение данным обработчиком. Но иногда этой информации обработчику исключения оказывается недостаточно, и тогда используют поля, собранные в структуру CONTEXT.

Например, если исключение не обрабатывается, а управление возвращается обратно в точку генерации исключения (значение фильтрующего выражения равно EXCEPTION_CONTINUE_EXECUTION), то при возврате вновь возникнет то же исключение. Изменив соответствующие поля структуры CONTEXT, мы избегаем замкнутого круга, например:

```
static int xfilter(EXCEPTION_POINTERS *xp)
  { int rc;
    EXCEPTION_RECORD *xr = xp->ExceptionRecord;
    CONTEXT *xc = xp->Context;
    switch (xr->ExceptionCode)
      { case EXCEPTION_BREAKPOINT:
              ++xc->Eip;   /* в коде программы остались встроенные точки
          останова перешагнем через них, изменив адрес команды на 1байт */
            rc = EXCEPTION_CONTINUE_EXECUTION;
            break;
        case EXCEPTION_ACCESS_VIOLATION:
            rc = EXCEPTION_EXECUTE_HANDLER;
            break;
        default:      // продолжить поиск обработчика
            rc = EXCEPTION_CONTINUE_SEARCH;
            break;    };
    return rc;
  }
  ...
  EXCEPTION_POINTERS *xp;
  try {     func();   }
  __except(xfilter(xp = GetExceptionInformation())) { abort();   }
```

Для генерации исключения используется специальная функция

void RaiseException(DWORD <код исключения>, **DWORD** <флаг>,
DWORD <количество аргументов>,
const DWORD *<адрес массива 32-разрядных аргументов>);

где <флаг> может принимать значения:

EXCEPTION_CONTINUABLE – исключение возобновимо;

EXCEPTION_NONCONTINUABLE – исключение не возобновимо.

Например:

```
#include <excpt.h>
#define MY_EXCEPTION 0x0000FACE
void func()
  { RaiseException(MY_EXCEPTION,
                                    EXCEPTION_CONTINUABLE,0,0);}
DWORD ExceptionFilter(DWORD dwCode)
    { if (dwCode==MY_EXCEPTION)
        return EXCEPTION_EXECUTE_HANDLER;
      else return EXCEPTION_COUNTINUE_SEARCH; }
void somefunc()
    { try {func(); }
      __except(Exception Filter(GetExceptionCode())) {...}
    }
```

Структурное управление исключениями поддерживает также завершающую конструкцию, которая выполняется независимо от того, было ли обнаружено исключение при выполнении защищенного блока:

```
__try {{<защищенный блок> }
__finally {<завершающая обработка>}
```

Или, соответственно, в С++:

```
try {<защищенный блок> }
__finally {<завершающая обработка>}
```

Например:

```
try
  { float f = 1.0,   g = 0.0;
     try   { e = f / g; } // генерируется исключение «деление на нуль»
     __except(EXCEPTION_EXECUTE_HANDLER)
          { <обработка исключения> }
  }
     __finally { <завершающая обработка> }
```

При выполнении завершающей обработки, так же как и при обычной обработке, вызываются деструкторы созданных локальных объектов. То же самое происходит, если исключения остаются необработанными. Локальные объекты не уничтожаются только в том случае, если необработанным остается исключение Win32.

Совместное использование различных механизмов обработки исключений. Механизм структурного управления исключениями был создан при разработке операционной системы Windows NT, но включение библиотеки

excpt.h в C++ Builder позволяет использовать этот механизм при работе с исключениями Win32 с соблюдением некоторых правил:

1) исключения Win32 можно обрабатывать только try…__except (C++) или __try…__except (C) или соответственно try…__finally (C++) или __try… __finally (C); оператор catch эти исключения игнорирует;

2) неперехваченные исключения Win32 не обрабатываются функцией обработки неперехваченных исключений и функцией terminate(), а передаются операционной системе, что обычно приводит к аварийному завершению приложения;

3) обработчики исключений не получают копии объекта исключения, так как он не создается, а для получения информации об исключении используют функции GetExceptionCode () и GetExceptionInformation ().

При одновременной обработке исключений различных типов необходимо иметь в виду, что:

– исключения C++ не видимы для __except (блока обработки структурных исключений C), а исключения C не перехватываются catch;

– каждому блоку try может соответствовать один блок __except или последовательность блоков catch, и попытки нарушить это правило приводят к синтаксическим ошибкам.

Если возникает необходимость перехвата структурных исключений и исключений C++ для одной последовательности операторов, то соответствующие конструкции вкладываются одна в другую.

Пример 6.5. Совместная обработка исключений различных типов. Рассмотрим организацию совместной обработки исключения Win32 «Деление на ноль в вещественной арифметике» и исключения C++.

В файле Exception.h определим класс исключения и функцию, использующую это исключение:

```
#ifndef ExceptionH
#define ExceptionH
class MyException            // класс исключения
 { private:  char* what;  // поле сообщения
   public:      MyException(char* s);
                MyException(const MyException& e );
               ~MyException();
              char* msg()const;
 };
float func(float f,float g);
#endif
```

Тела методов и функции опишем в файле Exception.cpp:

```
include <vcl.h>
#pragma hdrstop
```

```
#include "Exception.h"
MyException::MyException(char* s = "Unknown"){ what = strdup(s); }
MyException::MyException(const MyException& e ){ what = strdup(e.what); }
MyException::~MyException(){delete[] what;       }
char* MyException::msg() const{ return what; }
float func(float f,float g)
 {float r=0;
  try
   {try
     { try  { r =f / g; }
       __except(EXCEPTION_EXECUTE_HANDLER)
          { throw(MyException("Ошибка Деление на нуль")); }
     }
    catch(const MyException& e)
     { ShowMessage(AnsiString(e.msg()));}
   }
  __finally { ShowMessage("Завершающая обработка"); }
  return r;
 }
#pragma package(smart_init)
```

Вызов функции можно осуществить следующим образом:

```
RezEdit->Text=FloatToStr(func(StrToFloat(DivEdit->Text),
                                StrToFloat(DvEdit->Text)));
```

Примечание. При отладке программ, использующих обработку исключений, следует иметь в виду, что среда C++ Builder, так же как и среда Delphi, фиксирует любые исключения, в том числе и обрабатываемые в программе. Получив сообщение среды, необходимо продолжить выполнение программы.

6.3. VCL-совместимые классы

Для поддержки библиотеки компонент VCL, реализованной в Delphi Pascal, были реализованы VCL-совместимые классы. Так же как и в Delphi Pascal, они наследуются от класса TObject. Для описания таких классов в базовую объектную модель C++ были добавлены следующие средства:

– возможность объявления опубликованных и OLE-совместимых секций в классе (**__published**, **__automated**);

– группа специальных модификаторов (**__declspec**);

– обеспечение совместимости по типам данных и параметров;

– совместимые средства обработки исключений.

Определение VCL-совместимого класса. На VCL-совместимые классы накладываются следующие ограничения:

1) при их объявлении не разрешается использовать множественное наследование (в том числе и виртуальные базовые классы);

2) объекты VCL-совместимых классов *должны создаваться динамически* с использованием оператора new, а уничтожаться delete;

3) эти классы обязательно должны иметь деструктор;

4) для таких классов компилятор автоматически не создает копирующего конструктора и не переопределяет оператор присваивания.

Определение VCL-совместимого класса в C++ Builder выглядит следующим образом:

```
class <имя объявляемого класса>:
                          <вид наследования> <имя родителя>
{ private:        <скрытые элементы класса>
  protected:      <защищенные элементы класса>
  public:         <общедоступные элементы класса>
  __published:    <опубликованные элементы класса>
  __automated:    <элементы, реализующие OLE-механизм>
};
```

По сравнению со стандартной моделью C++ при объявлении VCL-совместимого класса можно объявлять секции published и automated в соответствии с моделью, используемой Pascal Delphi.

Секция **_ _published** (см. раздел 5.1) используется для свойств, которые доступны через Инспектор объектов, если соответствующий класс включен в Палитру компонент. В остальном опубликованные свойства идентичны общедоступным. Единственное отличие опубликованных элементов от общедоступных заключается в том, что для опубликованных элементов дополнительно генерируется RTTI информация (информация о типе времени выполнения), которая позволяет приложению динамически запрашивать элементы данных, методы и свойства ранее неизвестных типов. Опубликованные свойства могут определяться только в классе, наследуемом от TObject.

В секции опубликованных элементов не разрешается объявлять конструкторы и деструкторы. Там допустимо описывать свойства, элементы, наследуемые от объявленных в VCL (методы и указатели на них). Причем описываемые поля должны быть объектными, а свойства – простыми (не свойствами-массивами и не индексными свойствами) типов: порядковые, вещественные, строки, множества, объекты или указатели на методы.

Секция **_ _automated** используется для описания элементов, реализующих OLE-механизм.

Описатель _ _declspec. Он используется в C++ Builder для вызова макроопределений поддержки VCL, описанных в sysdefs.h. Аргументы этого описателя рассмотрены ниже.

_ _declspec(delphiclass) – используется для описания классов, производных от TObject (пример 6.6).

__declspec(delphireturn) – предназначен для описания классов, созданных в C++Builder для поддержки встроенных типов Delphi Pascal, которые не имеют аналогов в C++, таких как Currency, AnsiString, Variant, TDateTime, Set, и используется только VCL. Посредством этого объявления классы помечаются как VCL-совместимые при использовании в качестве параметров и возвращаемых значений функций VCL. Этот модификатор необходим, когда некая структура «по значению» передается между Delphi Pascal и C++.

__declspec(dynamic) – используется только для объявления динамических функций (см. раздел 5.2) в классах, наследуемых от TObject.

Пример 6.6. Статические, виртуальные и динамические полиморфные методы. В классе class1 объявляются статический statfunc(), виртуальный virtfunc() и динамический dinfunc() полиморфные методы. Класс class2 перекрывает эти методы своими.

Описание классов выполнено в файле Object.h:

```
class __declspec(delphiclass) class1 : public TObject
{public:
   char* polyfunc();
   virtual char*  virtfunc();
   __declspec(dynamic) char* dynfunc();
};
class __declspec(delphiclass) class2 : public class1
{public:
   char* polyfunc();
   char* virtfunc();
   char* dynfunc();
};
```

Тела функций описаны в файле Object.cpp:

```
#include "Object.h"
char* class1::polyfunc(){ return "статический 1";}
char* class1::virtfunc(){ return "виртуальный 1"; }
char* class1::dynfunc() { return "динамический 1"; }
char* class2::polyfunc(){ return "статический 2";}
char* class2::virtfunc(){ return "виртуальный 2"; }
char* class2::dynfunc() { return "динамический 2"; }
```

Затем создается динамический объект типа class2 и его адрес присваивается указателю на базовый класс:

```
class2 * Class2 = new class2;
class1 * Class1 = Class2;
```

Теперь при обращении к каждой из трех функций будут получены следующие результаты:

Edit->Text=Class1->polyfunc(); // статический 1 (!!!)
Edit->Text=Class2->polyfunc(); // статический 2
Edit->Text=Class1->virtfunc(); // виртуальный 2
Edit->Text=Class2->virtfunc(); // виртуальный 2
Edit->Text=Class1->dynfunc(); // динамический 2
Edit->Text=Class2->dynfunc(); // динамический 2

Как и ожидалось, по указателю на базовый класс при раннем связывании (статический полиморфный метод) вызывается функция базового класса. Во всех остальных случаях вызов осуществлялся верно.

_ _declspec(hidesbase) – используется в C++, когда необходимо указать компилятору, что данная виртуальная функция, несмотря на то же имя и тот же список параметров, образует новое семейство виртуальных функций. Необходимость такого описания вызвана тем, что в Delphi Pascal перекрывающие аспекты виртуальных функций описываются override, в то время как повторное использование описания virtual для функции с тем же именем соответствует объявлению нового семейства виртуальных функций. Для отображения этого случая в C++ и используется описатель hidesbase.

Так, если в классе T1 описана виртуальная функция func без аргументов, а в классе T2 необходимо описать функцию с тем же именем и также без аргументов, которая никак не связана с func, то эта функция описывается HIDESBASE:

virtual void T1::func(void);
HIDESBASE void T2::func(void);

При отсутствии специального описателя функция T2::func() будет интерпретироваться как аспект виртуальной функции T1::func().

_ _declspec(package) – указывает, что код определения класса должен компилироваться в пакет. *Пакет* представляет собой DLL специального вида, используемую приложениями C++ Builder.

_ _declspec(pascalimplementation) – указывает, что код, определяющий класс, был реализован в Delphi Pascal. Этот описатель используется в заголовочных файлах с расширением .hpp, описывающих макросы совместимости с библиотекой VCL.

_ _declspec (dllimport) – применяется для описания прототипов функций DLL.

Для статического добавления DLL к приложению C++Builder необходимо в файле проекта .bpr приложения внести имя импортируемой библиотеки в список библиотек, ассоциированный с переменной ALLLIB. При необходимости путь к этой библиотеке указывается в списке L опции

переменной LFLAGS (опции компоновщика). После этого экспортируемые функции DLL становятся доступны приложению. Прототипы используемых функций DLL предваряются описателем _ _declspec(dllimport), например:

_ _declspec(dllimport)
<тип результата> <имя импортируемой функции>(<параметры>);

Для динамической загрузки DLL во время работы приложения используется функция Windows API **LoadLibrary()** и функция **GetProcAddress()** – для получения адреса конкретной функции.

Совместимость по типам данных. В связи с тем, что некоторые типы по-разному определены в Delphi Pascal и C++, возможно появление трудноуловимых ошибок, вызванных различием в определении типов.

Булевы типы. Значение «истина» для типов Delphi Pascal ByteBool, WordBool, и LongBool представляется минус единицей, «ложь» – нулем. Значения для типа Boolean остались традиционными «истина» – единица, а «ложь» – ноль. C++ правильно интерпретирует эти значения. Проблема возникает, если функция WinAPI или другая функция возвращает результат типа BOOL (в котором «истина» кодируется единицей). Сравнение результата с переменными указанных типов произойдет только, если оба сравниваемых значения равны 0. Если необходимо выделить ситуации совпадения значений, то такое сравнение следует программировать как !A = !B:

A:ByteBool	*BOOL B*	*A=B*	*!A=!B*
0 (False)	0 (False)	0=0 (True)	!0 = !0 (True)
0 (False)	1 (True)	0=1 (False)	!0 = !1 (False)
–1 (True)	0 (False)	–1=0 (False)	!–1 = !0 (False)
–1 (True)	1 (True)	–1=1 (**False!**)	!–1 = !1 (True)

Символьные типы. В C++ символьный тип – знаковый, а в Delphi Pascal – беззнаковый. Вероятность появления ошибок, вызванных этим различием, невелика.

Открытые массивы. Для обеспечения независимости процедур и функций с параметрами-массивами от размера этих массивов в Pascal используются «открытые массивы». В аналогичных случаях C++ передает адрес массива и значение последнего индекса (n-1).

Так, описанию

function Mean(Data: array of Double): Extended;

в C++ будет соответствовать описание

*Extended _ _fastcall Mean(const double * Data, const int Data_Size);*

Вызов этой функции в C++ можно выполнить одним из следующих способов:

double d[] = { 3.1, 4.4, 5.6 }; long double x = Mean(d, 2);
или
long double y = Mean(d, (sizeof(d) / sizeof(d[0])) - 1);
или
long double z = Mean(d, ARRAYSIZE(d) - 1);

В последнем случае используется макрос ARRAYSIZE, определенный в sysdefs.h.

R T T I. Delphi Pascal содержит конструкции, использующие RTTI (см. раздел 5.4). C++Builder предлагает в аналогичных случаях свои конструкции:

Delphi Pascal RTTI	C++ RTTI
Динамическая проверка типа объекта	
*if Sender **is** TButton...* {is при неудаче возвращает false, а при удаче – truu}	*if (**dynamic_cast** <TButton*> (Sender) /** dynamic_cast при неудаче возвращает NULL, а при удаче – указатель */
Динамическое переопределение типа объекта	
*b := Sender **as** TButton;* {при неудаче генерируется исключение}	*TButton& ref_b = **dynamic_cast** <TButton&> (*Sender);* /* при неудаче генерируется исключение*/
Динамическое определение типа объекта	
*Sender.**ClassName***	***typeid**(*Sender).**name**();*

Метод TObject.ClassName, который возвращает строку, содержащую имя реального типа объекта независимо от типа используемой переменной, имеет аналог в C++ - name(). Остальные методы аналогов не имеют, но они описаны в TObject как общедоступные и потому могут вызываться напрямую (см. раздел 5.4).

Обработка исключений VCL-совместимых классов. VCL-совместимые классы используют механизм обработки исключений Delphi Pascal. Стандартная обработка предполагает, как правило, вывод сообщений об ошибках.

C++ Builder включает классы для автоматической обработки таких исключительных ситуаций, как деление на нуль, ошибки операций с файлами, неверное преобразование типа и т.п. Все эти классы наследуются от класса Exception (см. раздел 5.8). Для перехвата этих исключений используется конструкция C++:

catch (<класс исключения> **&**<переменная>)

Переменная, как это и принято в C++, используется для получения

значений полей класса и вызова его методов, например:

```
void __fastcall TForm1::ThrowException(TObject *Sender)
{ try {   throw Exception("VCL component"); }
  catch(const Exception &E)
    { ShowMessage(AnsiString(E.ClassName())+ E.Message); }
}
```

В данном примере генерируется исключение, которое тут же перехватывается и обрабатывается.

Наиболее часто используемые классы исключений перечислены в разделе 5.8.

При необходимости программист может создать собственный класс-исключение, например, наследуя его от одного из существующих.

Необходимо хорошо представлять себе различие между исключениями C++ и исключениями VCL:

1. Если при конструировании объекта возникает исключение, то в C++ деструкторы вызываются только для полей и базовых классов, которые были полностью сконструированы, а в VCL – в том числе и для объекта, при конструировании которого обнаружено исключение.

2. В C++ исключения могут перехватываться по ссылке, указателю или значению, а в VCL – только по ссылке или по значению. Попытки перехвата по значению приводят к синтаксической ошибке. Исключения от схем контроля или операционной системы, такие как EAccessViolation, могут перехватываться только по ссылке.

3. Для исключений, перехваченных по ссылке, нельзя повторно генерировать throw. Последнее связано с тем, что как только исключение операционной системы или VCL распознается как исключение C++, оно уже не может быть повторено в своем качестве из блока catch.

4. Программист не должен освобождать память объекта исключения VCL после обработки исключения.

5. Генерации исключений VCL выполняются «по значению».

Для обработки исключений операционной системы, таких как ошибки арифметики, переполнение стека, нарушение правил доступа и т.д., используется специальная предобработка и преобразование их к исключениям VCL. Т.е. VCL нормально обрабатывает эти исключения:

```
try{ char * p = 0;
      *p = 0;}
catch (const EAccessViolation &e)
          {<обработка исключения>}
```

В качестве примера разработки VCL-совместимого класса рассмотрим проектирование главного окна приложения.

Пример 6.7. Разработка VCL-совместимого класса для реализации главного окна приложения «Динамический массив». Главное окно приложения «Динамический массив» должно обеспечивать возможность тестирования всех предусмотренных над множеством операций (рис. 6.2).

В процессе визуального проектирования C++ Builder автоматически строит описание класса TMainForm, куда добавляются поля-указатели на визуальные компоненты и прототипы методов обработки событий, используемых программистом для реализации данного приложения. Это описание помещается в файл Main.h:

```
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
class TMainForm : public TForm
{
__published:          // опубликованные компоненты класса
    TLabel *MaxSizeLabel; // метка Максимальный размер массива
    TEdit *MaxSizeEdit;   // редактор Максимальный размер массива
    TBevel *Bevel1;       // рамка
    TButton *ModifyButton; // кнопка Изменить
    TButton *InsertButton; // кнопка Вставить
    TButton *DeleteButton; // кнопка Удалить
    TButton *DataButton;   // кнопка Изменить данные
    TButton *ExitButton;   // кнопка Выход
    TStringGrid *DataStringGrid; // таблица для отображения вектора
    TLabel *IndexLabel;    // метка Индекс
    TLabel *ValueLabel;    // метка Значение
```

Рис. 6.2. Главное окно приложения «Динамический массив»

```
    TEdit *IndexEdit;          // редактор индекса
    TEdit *ValueEdit;          // редактор значения
    TLabel *CommentLabel; // метка Комментарий
    void __fastcall ExitButtonClick(TObject Sender); /* обработчик события
                                        "Нажатие на кнопку Выход" */
    void __fastcall ModifyButtonClick(TObject *Sender); /* обработчик
                              события "Нажатие на кнопку Изменить" */
    void __fastcall DataStringGridKeyPress(TObject *Sender, char &Key); /*
                              обработчик события "Ввод символа" */
    void __fastcall InsertButtonClick(TObject *Sender); /* обработчик
                              события "Нажатие на кнопку Вставить" */
    void __fastcall DeleteButtonClick(TObject *Sender); /* обработчик
                              события "Нажатие на кнопку Удалить" */
    void __fastcall DataButtonClick(TObject *Sender); /* обработчик
                    события "Нажатие на кнопку Изменить данные" */
    void __fastcall FormActivate(TObject *Sender); /* обработчик события
                                   "Активация формы" */
private:    // внутренние компоненты класса
public:           // общедоступные компоненты класса
    __fastcall TMainForm(TComponent* Owner); // конструктор
    __fastcall ~TMainForm();                 // деструктор
};
extern PACKAGE TMainForm *MainForm;
#endif
```

Тела обработчиков событий программируются в файле Main.cpp. Наиболее интересные фрагменты текста программы выделены (работа с множествами Delphi Pascal, обработка исключений различных типов, проверка кода нажатой клавиши, работа со строками AnsiString, динамическая проверка типа и т.п.):

```
#include <vcl.h>
#pragma hdrstop
#include "Array.h"
#include "Main.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
TMasByte* A;
__fastcall TMainForm::TMainForm(TComponent* Owner) : TForm(Owner)
    { A=new TMasByte(10);}
__fastcall TMainForm::~TMainForm() { delete A; }
void __fastcall TMainForm::ExitButtonClick(TObject *Sender) { Close(); }
void __fastcall TMainForm::ModifyButtonClick(TObject *Sender)
```

```
{ short Ind;    unsigned char Value;   AnsiString Num("индекса");
  TMsgDlgButtons Set1;    Set1<<mbOK;  // объявить множество
 try {Ind=StrToInt(IndexEdit->Text); /* выполнить, контролируя
                                                исключения */
       Num="элемента";
       Value=StrToInt(ValueEdit->Text);
       A->Modify(Ind,Value);
       A->OutputMas(DataStringGrid,0,0);   }
  catch (EConvertError&)  /* перехватить исключение "Ошибка
                                                преобразования" */
    { AnsiString s="Неверно введено значение ";
      MessageDlg(s+Num,mtInformation,Set1,0);  }
  catch (char * Mes)   /* перехватить исключения от операций над
                                           динамическим массивом */
    { MessageDlg(Mes,mtInformation,Set1,0);   }
}
void __fastcall TMainForm::DataStringGridKeyPress(TObject *Sender,
                                                        char &Key)
{  if (Key==VK_RETURN)    // если нажата клавиша Enter
     { Key=0;
      try {A->InputMas(DataStringGrid,0,0);
          TGridOptions Set1;                 // объявить множество
          Set1=DataStringGrid->Options;
          Set1>>goEditing>>goAlwaysShowEditor>>goTabs;
          DataStringGrid->Options=Set1;
          DataStringGrid->Enabled=false;
          ModifyButton->Enabled=true;
          InsertButton->Enabled=true;
          DeleteButton->Enabled=true;
          DataButton->Enabled=true;
          IndexEdit->SetFocus();
          DataStringGrid->Col=0;
         CommentLabel->Visible=false;    }
      catch (char* Mes)
       { TMsgDlgButtons Set2;             // объявить множество
         Set2<<mbOK;
         MessageDlg(Mes,mtInformation,Set2,0);     }
     }
}
void __fastcall TMainForm::InsertButtonClick(TObject *Sender)
{ short Ind;
  unsigned char Value;
```

```
  AnsiString Num("индекса");
  TMsgDlgButtons Set1;
  Set1<<mbOK;
  try {Ind=StrToInt(IndexEdit->Text); /* выполнить, контролируя
                                                     исключния */
       Num="элемента";
       Value=StrToInt(ValueEdit->Text);
       A->Insert(Ind,Value);
       A->OutputMas(DataStringGrid,0,0);    }
  catch (EConvertError&)      /* перехватить исключение "Ошибка
                                                     преобразования" */
    { AnsiString s="Неверно введено значение ";
      MessageDlg(s+Num,mtInformation,Set1,0);    }
  catch (char * Mes)    /* перехватить исключения от операций над
                                                     динамическим массивом */
    { MessageDlg(Mes,mtInformation,Set1,0); }
}
void __fastcall TMainForm::DeleteButtonClick(TObject *Sender)
{ short Ind;  TMsgDlgButtons Set1;
  Set1<<mbOK;
  try {Ind=StrToInt(IndexEdit->Text);
       A->Delete(Ind);
       A->OutputMas(DataStringGrid,0,0);   }
  catch (EConvertError&)
    { MessageDlg("Неверно введено значение индекса.", mtInformation, Set1,0);
                                                                        }
  catch (char * Mes)
    { MessageDlg(Mes,mtInformation,Set1,0);    }
}
void __fastcall TMainForm::DataButtonClick(TObject *Sender)
  { FormActivate(DataButton); }
void __fastcall TMainForm::FormActivate(TObject *Sender)
{ CommentLabel->Visible=true;
  if (dynamic_cast<TButton*>(Sender)) // если отправитель – кнопка, то
    { for (int i=0;i<10;i++) DataStringGrid->Cells[i][0]="";
      TGridOptions Set1;      Set1=DataStringGrid->Options;
      Set1<<goEditing<<goAlwaysShowEditor<<goTabs;
      DataStringGrid->Options=Set1;
      DataStringGrid->Enabled=true;
      DataStringGrid->Col=0;       DataStringGrid->SetFocus();
      delete A;          A=new TMasByte(10); /* пересоздать вектор */   }
}
```

6.4. Различие реализации объектных моделей C++, Delphi и C++ Builder

При создании программного обеспечения с использованием C++ Builder необходимо учитывать различие реализации объектных моделей C++, Delphi и C++ Builder.

Изначально Pascal и C++ использовали различные реализации объектных моделей в части конструирования объектов и их уничтожения. Для поддержки библиотеки VCL в C++ Builder разработчики среды вынуждены были реализовать не только механизмы C++, но и механизмы Delphi. Рассмотрим основные различия объектных моделей.

Порядок конструирования объектов.

C + +. Порядок конструирования объектов в C++ определяется следующим образом:

- виртуальные базовые классы;
- прочие базовые классы;
- производные классы.

Причем в каждый момент времени класс конструируемого объекта *совпадает с классом выполняющегося конструктора.* Соответственно, если конструктор вызывает виртуальный метод, то вызывается аспект, соответствующий текущему классу.

Borland Pascal 7.0 и Delphi. В Delphi Pascal автоматически вызывается только конструктор класса, объект которого создается, хотя память выделяется и под поля базовых классов. Для вызова конструкторов базовых классов программист должен использовать специальный оператор inherited (см. раздел 5.1), указывая его в конструкторе каждого производного класса. По соглашению этот оператор используется для вызова всех существующих базовых конструкторов. Тип объекта устанавливается сразу и не меняется при вызове конструкторов базовых классов. Аспект виртуального метода, вызываемого из конструктора, таким образом, будет определяться классом создаваемого объекта и не будет меняться в процессе конструирования.

C + + Builder. Объекты VCL-совместимых классов в C++ Builder конструируются по правилам Delphi Pascal, но используют синтаксис C++. Это означает, что вызов конструкторов всех базовых не VCL-совместимых классов осуществляется через список инициализации. В списке инициализации также указывается вызов конструктора ближайшего класса VCL. Этот класс будет обрабатываться первым и в процессе создания объекта вызовет конструкторы остальных классов VCL, используя inherited. Затем будут вызваны конструкторы классов, описанных в C++, начиная с класса, наследуемого непосредственно от VCL. Тип объекта во время конструирования и диспетчирования виртуальных методов выполняется по правилам Delphi Pascal.

Объекты обычных классов конструируются по правилам C++.

На рис. 6.3 показано, как происходит конструирование объектов класса, наследуемого от VCL.

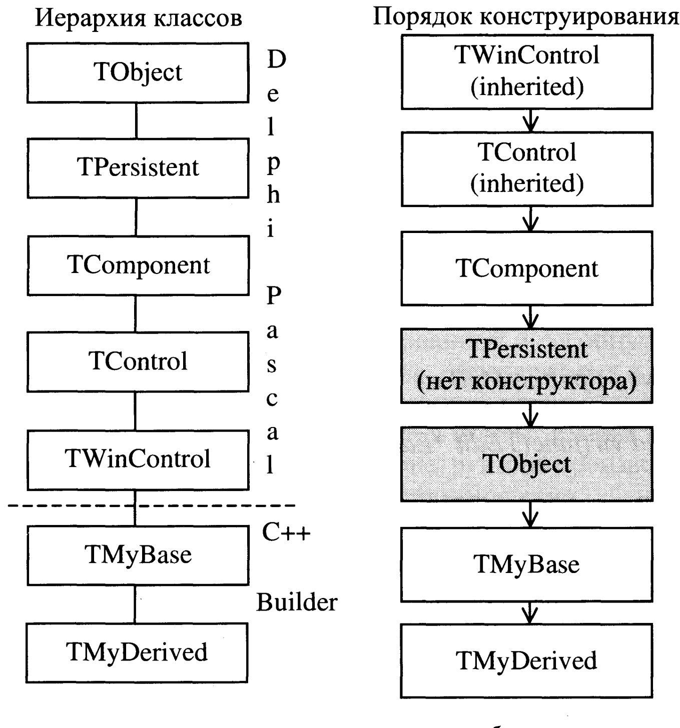


Рис. 6.3. Иерархия классов и порядок конструирования классов, наследуемых от классов VCL

Класс MyDerived наследуется от MyBase, производного от класса VCL TWinControl. Классы MyDerived и MyBase создаются в С++ Builder. Класс TWinControl описан в библиотеке VCL, т.е. на Delphi Pascal.

Примечание. Конструктор класса TComponent не содержит оператора inherited, так как класс TPersistent не имеет конструктора. Класс TObject включает пустой конструктор, который не вызывается.

Реализация вызова виртуальных методов из конструкторов. По правилам С++ вызов виртуальных методов диспетчируется в соответствии с текущим типом объекта. Соответственно, аспект виртуального метода, вызываемого из конструктора обычного класса С++, зависит от этапа конструирования, на котором вызывается метод. Для VCL-совместимых классов, поскольку для них тип конструируемого объекта устанавливается сразу, вызываемый аспект виртуального метода всегда соответствует типу конструируемого объекта.

В примере 6.8 сравнивается перекрытие виртуальных методов в классах С++ и классах VCL. Классы MyBase и MyDerived определены в стиле С++. Классы MyVCLBase и MyVCLDerived наследуются от TObject (стиль VCL).

Виртуальный метод virtfunc () перекрывается в обоих производных классах, но вызывается только в конструкторах базовых классов.

Пример 6.8. Перекрытие виртуальных методов в C++ классах и VCL-совместимых классах. Опишем две иерархии классов: одну в стиле обычного C++, вторую – в стиле VCL:

```
class MyBase      // не VCL класс
{ public:    MyBase(TEdit *Edit);  // конструктор
             virtual void virtfunc(TEdit *Edit);
};
class MyDerived : public MyBase
{public:  MyDerived(TEdit *Edit);
    virtual void virtfunc(TEdit *Edit);
};
class MyVCLBase : public TObject        // класс в стиле VCL
{ public:
    __fastcall MyVCLBase(TEdit *Edit);
    virtual void __fastcall virtfunc(TEdit *Edit);
};
class MyVCLDerived : public MyVCLBase
{ public:
    __fastcall MyVCLDerived(TEdit *Edit);
    virtual void __fastcall virtfunc(TEdit *Edit);
};
```

В конструкторах базовых классов предусмотрим вызов виртуального метода:

```
MyBase::MyBase(TEdit *Edit) { virtfunc(Edit); }                // вызов
void MyBase::virtfunc(TEdit *Edit){Edit->Text="Метод базы";}
MyDerived::MyDerived(TEdit *Edit):MyBase(Edit){}
void MyDerived::virtfunc(TEdit *Edit)
                { Edit->Text="Метод прозводного класса";}
__fastcall MyVCLBase::MyVCLBase(TEdit *Edit) {virtfunc(Edit);} // вызов
void __fastcall MyVCLBase::virtfunc(TEdit *Edit)
                { Edit->Text="Метод базы";}
__fastcall MyVCLDerived::MyVCLDerived(TEdit *Edit):MyVCLBase(Edit){}
void __fastcall MyVCLDerived::virtfunc(TEdit *Edit)
                { Edit->Text="Метод прозводного класса";}
```

Объявление переменных производных классов обеих иерархий приведет к автоматическому вызову конструкторов, при выполнении которых будут вызваны виртуальные методы:

```
MyDerived d(Edit1);          // выведет: Метод базы
MyVCLDerived *pvd = new MyVCLDerived(Edit2);
                                        // выведет: Метод производного класса
```

Такой результат объясняется тем, что в момент конструирования базового класса тип объекта в первом случае совпадал с базовым, как принято в модели C++, а во втором случае – был установлен сразу. Соответственно, и виртуальный метод в первом случае вызывался для базового класса, а во втором – для производного.

Последовательность и правила инициализация полей данных объекта. В Delphi Pascal все неинициализированные поля обнуляются. То же самое происходит и с полями объектов классов, определяемых в стиле VCL. В C++ это не принято. Все требуемые поля инициализируются через список инициализации конструктора. Следовательно, в момент вызова конструктора базового класса поля, инициализируемые в конструкторе производного класса или его списке инициализации, еще не получили значения.

Пример 6.9. Инициализация полей при конструировании объектов VCL- совместимых классов. Опишем иерархию VCL-совместимых классов:

```
class Base : public TObject
{ public:  __fastcall Base();
        virtual void __fastcall init();
};
class Derived : public Base
{ public:  __fastcall Derived(int nz);
        virtual void __fastcall init();
 private:  int not_zero;
};
```

Определим действия конструктора объектов данных классов:

```
__fastcall Base::Base() { init(); }
void __fastcall Base::init() { }
__fastcall Derived::Derived(int nz) : not_zero(nz) { }
void __fastcall Derived::init()
        {if (not_zero == 0) throw Exception("поле not_zero обнулено!");}
```

Теперь попробуем вызвать конструктор и проверить инициализацию полей:

```
Derived *d42 = new Derived(42); // генерируется исключение
```

Исключение генерируется, поскольку Base конструируется перед Derived и, следовательно, поле not_zero обнулено по правилам Delphi Pascal. Значение

данному полю будет присвоено позднее, когда будет выполняться список инициализации.

Вызов деструктора при возникновении исключения в конструкторе. C++ Builder использует два различных механизма уничтожения объектов: используемый Delphi Pascal и используемый C++. В первом случае при возникновении исключения в процессе конструирования вызывается деструктор. Во втором – вызываются виртуальные методы из деструктора. VCL-совместимые классы используют механизмы обоих языков.

Рассмотрим пример. Пусть класс C наследуется от B, а B, в свою очередь – от A:

```
class A           {...};
class B: public A {...};
class C: public B {...};
```

Если при выполнении конструктора класса B во время конструирования объекта класса C возникнет исключение, то произойдет следующее.

В C++ сначала будут вызваны деструкторы всех уже сконструированных элементов класса B, затем будут вызваны деструктор A и деструкторы его элементов. Деструкторы B и C не вызываются.

В Delphi Pascal только деструктор уничтожаемого объекта вызывается автоматически. Для рассматриваемого примера это деструктор C. Для вызова остальных деструкторов программист должен использовать оператор inherited. В данном примере, если деструктор C вызывает, как положено, деструктор B, а деструктор B в свою очередь – деструктор A, то деструкторы B и A будут вызваны в названном порядке. Независимо от того, вызывался ли уже конструктор A до возникновения исключения, деструктор A будет вызван из деструктора B. Более того, поскольку конструкторы базовых классов обычно вызываются в начале конструктора производного класса, очень важно, что и деструктор C вызывается до завершения выполнения его конструктора.

В C++Builder классы, определенные в VCL, описанные на Delphi Pascal, используют способ вызова деструктора, принятый в Delphi Pascal.

VCL-совместимые классы, реализуемые в C++Builder, не используют строго один из двух методов. Это выражается в том, что все деструкторы вызываются, но тела тех, которые не должны активизироваться по правилам C++, не выполняются.

Вызов виртуальных методов из деструкторов. Вызов виртуальных методов из деструкторов выполняется по той же схеме, что и вызов из конструкторов. Для классов, определенных в стиле VCL, уничтожение начинается с элементов производного класса, затем уничтожаются элементы базового класса и т.д. Тип объекта сохраняется на всех этапах процесса уничтожения, следовательно, в деструкторах всегда вызываются виртуальные методы класса, которому принадлежит уничтожаемый объект. TObject содержит два виртуальных метода: BeforeDestruction и AfterConstruction, использование которых позволяет программисту писать код, который

выполняется до и после уничтожения объектов соответственно. AfterConstruction вызывается после вызова последнего конструктора, а BeforeDestruction – перед вызовом первого деструктора. Эти методы объявлены общедоступными и вызываются автоматически.

Пример 6.10. Вызов виртуального метода из конструктора. Разработать программу, которая строит график функции по формуле, введенной пользователем. Реализовать основные математические операции кроме унарного минуса: сложение, вычитание, умножение, деление и возможность использования скобок для изменения порядка выполнения указанных операций. Аргумент обозначать символом x, функцию – символом y. Функция строится на отрезке, определенном пользователем, и с заданным шагом. Предусмотреть возможность построения графика для другого диапазона изменения аргумента или с другим шагом, а также возможность изменения самой функции. Обеспечить возможность рисования графика в рамке и без рамки.

Условие задачи определяет внешний вид интерфейса (рис. 6. 4).

В результате объектной декомпозиции получаем, что приложение должно включать следующие объекты: Окно приложения, График, График с рамкой и Функция. Окно приложения создает либо График, либо График с рамкой. Объект Функция вызывается из Графиков для построения бинарного дерева разбора выражения и вычисления значения функции при заданном аргументе по этому дереву.

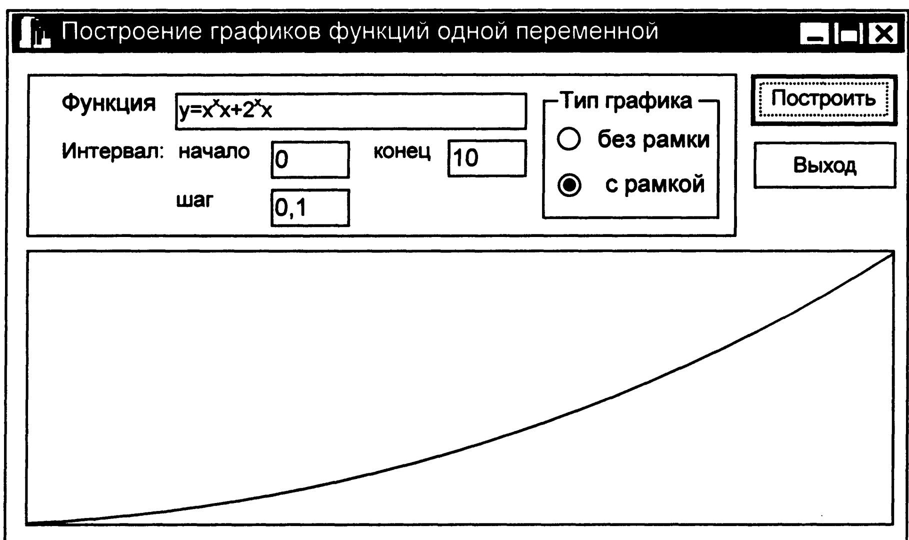


Рис. 6.4. Внешний вид окна программы

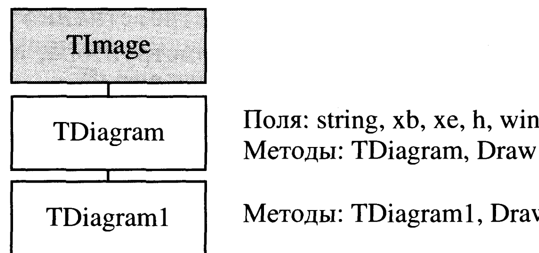


Рис. 6.5. Иерархия классов TImage-TDiagram-TDiagram 1

Далее проектируем классы. Класс TMainForm наследуем от TForm, добавив в него объектные поля интерфейсных элементов. Класс TFunction (функция) будем разрабатывать как самостоятельный класс. Класс TDiagram (график) наследуем от TImage, а класс TDiagram1 (график с рамкой) – от TDiagram, переопределив метод рисования графика Draw (рис. 6.5). Метод Draw будем вызывать из конструктора базового класса TDiagram, следовательно, необходимо позднее связывание, значит данный метод объявляем виртуальным. Классы TDiagram и TDiagram1 наследуем от TImage, т.е. эти классы VCL-совместимые, следовательно, никаких проблем с вызовом виртуальных методов из конструктора не будет.

Реализацию начнем с интерфейса. После определения компонент интерфейса на графе состояний интерфейса отмечаем события, обработчики которых реализуют данное приложение (рис. 6.6).

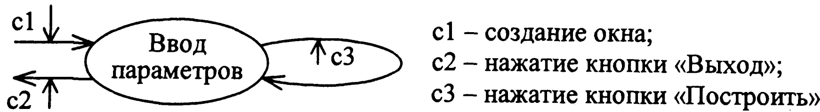


Рис. 6.6. Граф состояний интерфейса

Ниже приведены тексты модулей программы.

Модуль Main (описания класса TMainForm) :

1. Заголовок Main.h

```
#ifndef MainH
#define MainH
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
class TMainForm : public TForm
{__published:      // IDE-managed Components
    TButton *BuildButton; TButton *ExitButton;
    TPanel *Panel; TPanel *ImagePanel;
    TLabel *FLabel; TEdit *FEdit;
```

```
        TLabel *BeginLabel; TEdit *BeginEdit;
        TLabel *EndLabel; TEdit *EndEdit;
        TLabel *StepLabel; TEdit *StepEdit;
        TRadioGroup *TypeRadioGroup;
        void __fastcall ExitButtonClick(TObject *Sender);
        void __fastcall BuildButtonClick(TObject *Sender);
public:     __fastcall TMainForm(TComponent* Owner);
};
extern PACKAGE TMainForm *MainForm;
#endif
```

2. Описание методов Main.cpp

```
#include <vcl.h>
#pragma hdrstop
#include "Main.h"
#include <SysUtils.hpp>
#include "Grafic.h"
#include <vcl/dstring.h>
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
TDiagram *r=NULL;
__fastcall TMainForm::TMainForm(TComponent* Owner) :
                                                    TForm(Owner) {}
void __fastcall TMainForm::ExitButtonClick(TObject *Sender)
{ if (r!=NULL) delete r;    Close();}
void __fastcall TMainForm::BuildButtonClick(TObject *Sender)
  { if (r!=NULL) delete r;    // если График существует – уничтожить
    float x1=(float)StrToFloat(BeginEdit->Text),
          x2=(float)StrToFloat(EndEdit->Text),
          h = (float)StrToFloat(StepEdit->Text);
    if (TypeRadioGroup->ItemIndex==0)  // конструирование Графика
       r=new TDiagram(ImagePanel,FEdit->Text.c_str()+2,x1,x2,h);
    else r=new TDiagram1(ImagePanel,FEdit->Text.c_str()+2,x1,x2,h);
}
```

Модуль Grafic (описания классов TDiagram и TDiagram1) :

1. Заголовок Grafic.h

```
#ifndef graficH
#define graficH
#include "FunctionUnit.h"
class TDiagram:public TImage
```

```
  {private:  char *string;   float xb,xe,h;
   protected: TRect win;
         virtual void __fastcall Draw(char *st,float xb,float xe,float h);
   public:
   __fastcall TDiagram(TPanel *Panel,char *st,float xb,float xe,float h);
  };
class TDiagram1:public TDiagram
  {protected: virtual void __fastcall Draw(char *st,float xb,float xe,float h);
   public:
    __fastcall TDiagram1(TPanel *Panel,char *st,float xb,float xe,float h):
       TDiagram(Panel,st,xb,xe,h){}
  };
#endif
```

2. Описание методов Grafic.cpp

```
#include <vcl.h>
#include <extctrls.hpp>
#pragma hdrstop
#include "FunctionUnit.h"
#include "grafic.h"
__fastcall TDiagram::
    TDiagram(TPanel *Panel,char *st,float xb,float xe,float h):TImage(Panel)
       { Align = alClient;  Parent=Panel;
         win=Rect(0,0,Width,Height);
         Draw(st,xb,xe,h);
       }
void __fastcall TDiagram::Draw(char *st,float xb,float xe,float h)
 { float *x,*y,ymin=1e30,ymax=-1e30,mx,my;   int i,k,*kx,*ky; bool key;
   k=(xe-xb)/h+1.5;
   x=new float[k]; y=new float[k];    kx=new int[k]; ky=new int[k];
   MyFunction F(st);
   for(i=0;i<k;i++)  // вычисляем значения аргумента и функции
     {x[i]=xb+i*h;      key=true; y[i]=F.Count(x[i],key);
      if ((y[i]>ymax)&key) ymax=y[i];      if ((y[i]<ymin)&key) ymin=y[i];
     }
   mx=win.Width()/(x[k-1]-x[0]); my=win.Height()/(ymax-ymin);
   for(i=0;i<k;i++)
     {kx[i]=(x[i]-x[0])*mx+win.Left;ky[i]=(ymax-y[i])*my+win.Top;}
   Canvas->Brush->Color=clWhite;  Canvas->Pen->Color=clBlack;
   Canvas->MoveTo(kx[0],ky[0]);
   for(i=1;i<k;i++)Canvas->LineTo(kx[i],ky[i]); // строим график
   delete[] x; delete[] y; delete[] kx; delete[] ky;
```

```
 }
void __fastcall TDiagram1::Draw(char *st,float xb,float xe,float h)
 { Canvas->Rectangle(win.Left,win.Top,win.Right,win.Bottom);
   TDiagram::Draw(st,xb,xe,h); //вызов родительского метода
 }
#pragma package(smart_init)
```

Модуль FunctionUnit (описание класса MyFunction) :

1. Заголовок FunctionUnit.h

```
#ifndef FunctionUnitH
#define FunctionUnitH
struct Top { char Operation;  float Value;  Top *left, *rigth; };
class MyFunction
 {private: char *string;
         Top *root;
         unsigned int MainOperation(char *st, char *Op);
         void BuildTree(Top *r,char *st);
         float FCount(Top *r,float x,bool &key);
         void Del(Top *r);
  public:
     MyFunction(char *formula);
     ~MyFunction();
     float Count(float x,bool key);
 };
#endif
```

2. Описание методов FunctionUnit.cpp

```
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <vcl.h>
#pragma hdrstop
#include "FunctionUnit.h"
unsigned int MyFunction::MainOperation(char *st, char *Op)
 { unsigned int j=0,k=0,i=0,p=0;
  while ((i<strlen(st))&(p==0))
     { if (st[i]=='(') j++;
       else if (st[i]==')') k++;
        else if ((j==k)&(strchr(Op,st[i])!=NULL))p=i;
       i++;
     }
  return p;
```

```
  }
 MyFunction::MyFunction(char *formula)
  { root=new Top; string=new char[strlen(formula)+1];
   strcpy(string,formula);   BuildTree(root,formula); // строим дерево
  }
 void MyFunction::BuildTree(Top *r,char *st)
  { unsigned int po,n;      char stl[80],str[80];
   po=MainOperation(st,"+-"); // ищем разделяющие операции «+» или «-»
   if (po==0) po=MainOperation(st,"*/"); // иначе «*» или «/»
   if (po!=0) // если разделяющий знак найден, то строим вершину
    { r->Operation=st[po];
     strncpy(stl,st,po); stl[po]='\0';
     if ((stl[0]=='(')&(MainOperation(stl,"+-*/")==0)) // снимаем скобки
                         {strncpy(stl,st+1,po-2); stl[po-2]='\0';}
     n=strlen(st)-po-1;
     strncpy(str,st+po+1,n);     str[n]='\0';
     if ((str[0]=='(')&(MainOperation(str,'+-*/")==0)) // снимаем скобки
                         {strncpy(str,st+1,po-2); str[po-2]='\0';}
     r->left=new(Top); BuildTree(r->left,stl); // строим левое поддерево
     r->rigth=new(Top);BuildTree(r->rigth,str);//строим правое поддерево
    }
   else { r->left=NULL;r->rigth=NULL;
        if (st[0]=='x') r->Operation='x';
        else {r->Operation='o';r->Value=atof(st);}
      }
  }
 void MyFunction::Del(Top *r)
  { if (r->rigth!=NULL) Del(r->rigth);
   if (r->left!=NULL) Del(r->left);
   delete(r);  }
 MyFunction::~MyFunction()
  { delete[]string;    if (root!=NULL)Del(root); }
 float MyFunction::FCount(Top *r,float x,bool &key)
  { if (!key) return 0;
   switch (r->Operation)
    {case 'o':return r->Value;
     case 'x':return x;
     case '+':return FCount(r->left,x,key)+FCount(r->rigth,x,key);
     case '-':return FCount(r->left,x,key)-FCount(r->rigth,x,key);
     case '*':return FCount(r->left,x,key)*FCount(r->rigth,x,key);
     case '/': try {return FCount(r->left,x,key)/FCount(r->rigth,x,key);}
          __except (EXCEPTION_EXECUTE_HANDLER){key=false;return 0;}
     default: {key=false; return 0;}
```

```
    }
  }
float MyFunction::Count(float x,bool key)
  { return FCount(root,x,key);}
#pragma package(smart_init)
```

Вопросы для самоконтроля

1. Какие два типа классов реализованы в C++Builder и почему? В каких случаях необходимо использовать каждый из них?

2. Какие средства были включены в базовую объектную модель C++? Как их можно использовать?

3. Почему в C++Builder три различных механизма обработки исключений? Расскажите о каждом из них. В каких случаях они используется ? Возможно ли их совместное применение?

4. Какие возможности реализованы в VCL-совместимых классах? Перечислите их и поясните, когда они могут быть использованы.

5. Назовите основные различия между механизмами реализации обычных и VCL-совместимых классов? Когда они проявляются?

6. Попробуйте самостоятельно (по аналогии с Delphi) создать приложение, которое генерирует сообщение и обрабатывает сообщение. Создайте в обработчике сообщения событие. Сравните полученную программу с программой примера 5.9. Поясните результаты.

ЗАКЛЮЧЕНИЕ

Существуют разные уровни программирования с применением ООП: можно просто использовать огромные библиотеки классов таких сред разработки программного обеспечения, как Delphy, C++ Builder или Visual C++, а можно изначально строить программу как дом, добавляя к ней все новые блоки – объекты, для реализации которых Вам придется создавать свои классы.

При написании прочитанного Вами учебника мы ставили перед собой цель максимально просто и одновременно подробно изложить современные представления как о самой технологии ООП, так и о средствах ее реализации, чтобы Вы могли более полно использовать предоставляемые ею возможности.

Конечно в рамках одной книги практически невозможно предусмотреть все ситуации и обсудить все нюансы разработки программ с использованием ООП. Кроме того, для практического освоения данной технологии Вам придется разработать несколько своих программ, постепенно изучая предлагаемые приемы программирования с использованием ООП и придумывая свои. Для уточнения сложных вопросов Вам многократно придется обращаться к другим книгам и читать справочную литературу. Однако мы надеемся, что, прочитав данную книгу, Вы получили достаточно целостное представление о возможностях ООП и областях его применения.

СПИСОК ЛИТЕРАТУРЫ

К главе 1

1. *Бадд Т.* Объектно-ориентированное программирование в действии: Пер. с англ. СПб.: Питер, 1997. 464 с.

2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на С++: Пер. с англ. М.: Бином, СПб.: Невский диалект, 1998. 560 с.

3. *Дал У., Дейкстра Э., К.Хоор.* Структурное программирование: Пер. с англ. М.: Мир, 1975. 247 с.

4. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию: Пер. с англ. М.: Мир, 1980. 278 с.

К главе 2

1. *Поляков Д.Б., Круглов И.Ю.* Программирование в среде Турбо-Паскаль (версия 5.5). М.: Изд-во МАИ, 1992. 576 с.

2. *Фаронов В.В.* Турбо-Паскаль в 3 кн. Кн. 1. Основы Турбо-Паскаля. М.: МВТУ – Фесто Дидактик, 1992. 304 с.

К главе 3

1. *Березин Б.И., Березин С.Б.* Начальный курс С и С++. М.: Диалог-МИФИ, 1997. 288 с.

2. *Вайнер Р., Пинсон Л.* С++ изнутри: Пер. с англ. Киев: ДиаСофт, 1993. 304 с.

3. *Лукас П.* С++ под рукой: Пер. с англ. Киев: ДиаСофт, 1993. 304 с.

4. *Подбельский В.В.* Язык Си++. М.: Финансы и статистика, 1995. 560 с.

5. *Скляров В.А.* Язык С++ и объектно-ориентированное программирование. М.: Высш. шк., 1997. 478 с.

6. *Страуструп Б.* Язык программирования С++.: Пер. с англ. СПб.; М.: Невский проспект – Изд-во БИНОМ, 1999. 991 с.

К главе 4

1. *Колверт Ч.* Программирование в Windows'95. Освой самостоятельно: Пер. с англ. М.: Вост. книжн. компания, 1996. 1008 с.

2. *Персон Р.* Windows 95 в подлиннике: Пер. с англ. СПб.: BHV-Санкт-Петербург, 1996. 736 с.

К главе 5

1. *Епашенников А.М., Епашенников В.А.*Программирование в среде Delphi: В 4-х ч. М.: Диалог-МИФИ, 1997–98.

2. *Лишер Р.* Секреты Delphi 2: Пер. с англ. Киев: НИПФ ДиаСофтЛтд., 1996. 800 с.

3. *Марченко А.И.* Программирование на языке Object Pascal 2.0. Киев: Юниор, 1998. 304 с.

4. *Мачто Д., Фолкнер Д.Р.* Delphi: Пер. с англ. М.: Бином, 1995. 464 с.

5. *Орлик С.В.* Секреты Delphi на примерах: Версии 1.0 и 2.0. М.: Бином, 1996. 316 с.

6. *Фаронов В.В.* Delphi 3. Учебный курс. М.: Нолидж, 1998. 400 с.

К главе 6

1. *Елманова Н.З., Кошель С.П.* Введение в Borland C++Builder. М.: Диалог-МИФИ, 1997. 272 с.

2. *Рейсдорф К.* Borland C++ Builder 3. Освой самостоятельно: Пер. с англ. М.: Бином, 1999. 736 с.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Абстракция 19, 21, 23

Декомпозиция объекта 28
объектная 25, 50, 196, 299
процедурная 11
Деструкторы виртуальные 76, 152
простые 38, 76, 110, 152
Дружественные функции 135

Инкапсуляция 20
Исключения 55, 251, 273
генерация 56, 254, 273
обработка 254, 274
обработка завершающая 252, 278

Классы абстрактные 133, 203
вложенные 108
иерархия 22, 35, 41, 68, 255
контейнерные 20, 63, 123, 205
локальные 52, 108, 161, 223
определение 35, 59, 99, 192
параметризованные 54, 154,
Композиция 45, 86
Конструкторы 38, 70, 110, 194, 295
копирующие 114, 126, 146, 159
неинициализирующие 113, 151
переопределяемые 110
по умолчанию 151

Метаклассы 48, 219
Методы абстрактные 133, 203, 144
виртуальные 69, 130, 202, 285
делегирование 49, 228, 263
динамические 202, 285
класса 222
перегруженные 207, 262
переопределенные 42, 67, 129, 162
полиморфные 42, 67, 130, 161
статические 109, 285
Модульность 18, 20

Наследование виртуальное 127, 137
множественное 40, 125
простое 40, 63, 118
свойств 18
Наполнение 44, 86

Объекты динамические 77, 85, 144
операции 34
переопределение типа 44, 266
поведение 33
полиморфные 41, 70
предметной области 29
простые 60
с динамическими полями 77, 144
состояние 32
статические 38, 144

Ограничение доступа 20, 37, 212
интерфейс 20, 37, 210
реализация 20, 38

Полиморфизм простой 41, 67, 129
сложный 42, 67, 129
чистый 42
Поля динамические 77, 144, 193
общедоступные 120, 139
объектные 86
статические 105

Программирование событийное 174
модульное 17
процедурное 11
структурное 12

Проектирование логическое 24, 93
физическое 24, 93

Свойства индексируемые 218, 272
массивы 214, 270
простые 210, 268
Связывание позднее 43
раннее 41

События 179
обработчики 180, 200
создание 246
Сообщения 33, 174
компонентов 247
методы обработки 174, 241
создание 241
цикл обработки 174, 249
Стек вызовов 56, 275

Типизация 21

Устойчивость 22

Шаблоны 42, 165

Языки объектно-ориентированные 22
сравнительные характеристики 23

RTTI 48, 220

TBM 44, 132, 220

ПЕРЕЧЕНЬ ПРИМЕРОВ

К главе 1

Пример 1.1. Процедурная декомпозиция (программа «Записная книжка») 14

Пример 1.2. Объектная декомпозиция (имитационная модель бензоколонки) 26

Пример 1.3. Декомпозиция объекта (Блок колонок) 28

Пример 1.4. Простейший графический редактор 29

Пример 1.5. Объектная декомпозия (программа «Записная книжка») 30

Пример 1.6. Описание класса (класс Окно) 36

Пример 1.7. Сокрытие реализации класса (класс Файл – продолжение примера 1.5) 38

Пример 1.8. Наследование (класс Окно_меняющее_цвет) 40

Пример 1.9. Простой полиморфизм (класс Окно_с_текстом) 42

Пример 1.10. Сложный полиморфизм 43

Пример 1.11. Композиция (класс Сообщение – продолжение примера 1.5) 46

Пример 1.12. Наполнение (класс Функция) 47

Пример 1.13. Делегирование методов (класс Фигура) 50

Пример 1.14. Контейнерный класс с итератором (класс Список) 53

Пример 1.15. Контейнерный класс с процедурой обработки всех объектов (класс Список) 54

Пример 1.16. Шаблон классов (шаблон классов Список) 55

К главе 2

Пример 2.1. Описание класса (класс Окно) 61

Пример 2.2. Разработка сложного класса без использования наследования (класс Символ) 64

Пример 2.3. Использование наследования (классы Окно и Символ) 66

Пример 2.4. Применение простого полиморфизма 68

Пример 2.5. Вызов виртуальных методов из методов базового класса 71

Пример 2.6. Использование процедуры с полиморфным объектом 72

Пример 2.7. Динамический объект с динамическим полем и контролем выделения памяти 77

Пример 2.8. Статический объект с динамическим полем и контролем выделения памяти 78

Пример 2.9. Использование динамических объектов (программа «Снежинки») 79

Пример 2.10. Размещение описания класса в модуле 84

Пример 2.11. Использование объектных полей 86

Пример 2.12. Использование полей – указателей на объекты 87

Пример 2.13. Программа «Текстовые эффекты» 92

К главе 3

Пример 3.1. Определение класса (класс Строка) 101

Пример 3.2. Различные способы инициализации полей объекта 103

Пример 3.3. Использование параметра this 105

Пример 3.4. Класс со статическими компонентами 107

Пример 3.5. Вложенные классы 108

Пример 3.6. Использование конструктора для инициализации полей класса 110

Пример 3.7. Использование переопределяемых конструкторов 111

Пример 3.8. Использование конструктора с аргументами по умолчанию 112

Пример 3.9. Использование конструктора со списком инициализации и неинициализирующего конструктора 113

Пример 3.10. Использование предполагаемого копирующего конструктора 115

Пример 3.11. Явное определение копирующего конструктора 116

Пример 3.12. Определение деструктора в классе 118

Пример 3.13. Описание производного класса с типом доступа public 119

Пример 3.14. Описание производного класса с видом наследования private 120

Пример 3.15. Порядок работы конструкторов базового и производного классов 122

Пример 3.16. Последовательность описания и вызова конструкторов и деструкторов при многоуровневой иерархии классов 123

Пример 3.17. Наследование от двух базовых классов 125

Пример 3.18. Виртуальное наследование 128

Пример 3.19. Использование раннего связывания 129

Пример 3.20. Использование виртуальных функций 131

Пример 3.21. Использование абстрактного класса при работе с полиморфными объектами 133

Пример 3.22. Внешняя дружественная функция 135

Пример 3.23. Дружественная функция – компонент другого класса 136

Пример 3.24. Объявление дружественного класса. 137

Пример 3.25. Описания функции-оператора вне класса 139

Пример 3.26. Пример описания компонентной функции-оператора 140

Пример 3.27. Переопределение коммутативной операции «умножение на скаляр» и операции «+» 141

Пример 3.28. Переопределение операций ввода – вывода 143

Пример 3.29. Конструирование и разрушение объектов с динамическими полями 145

Пример 3.30. Использование собственного копирующего конструктора 146

Пример 3.31. Использование простых динамических объектов 149

Пример 3.32. Обработка массива динамических объектов 150

Пример 3.33. Использование указателей на базовый класс и виртуального деструктора 152

Пример 3.34. Шаблон, позволяющий формировать одномерные динамические массивы из заданных элементов 154

Пример 3.35. Использование шаблонов для формирования массивов и печати их элементов 155

Пример 3.36. Использование шаблонов функций при создании шаблонов классов 156

Пример 3.37. Использование шаблона классов (шаблон классов «Множество») 157

Пример 3.38. Контейнерный класс с процедурой поэлементной обработки 161

Пример 3.39. Контейнер на основе шаблона 165

К главе 4

Пример 4.1. Приложение «Возведение чисел в квадрат» (вариант 1) 182

Пример 4.2. Приложение «Возведение чисел в квадрат» (вариант 2) 188

К главе 5

Пример 5.1. Определение класса (графический редактор «Окружности» – вариант 1) 196

Пример 5.2. Использование абстрактных методов (графический редактор «Окружности и квадраты» – вариант 1) 204

Пример 5.3. Использование простых свойств (графический редактор «Окружности» – вариант 2) 212

Пример 5.4. Использование свойств-массивов (класс «Динамический массив» – вариант 1) 215

Пример 5.5. Контейнер «Двусвязный линейный список» 223

Пример 5.6. Делегирование методов (графический редактор «Окружности и квадраты» – вариант 2) 229

Пример 5.7. Использование отношения «старший–младший» (приложение «Определение вида четырехугольника») 238

Пример 5.8. Передача/прием сообщения 244

Пример 5.9. Создание события 246

Пример 5.10. Прерывание длительной обработки 249

Пример 5.11. Использование исключений (класс «Динамический массив» – вариант 2) 256

К главе 6

Пример 6.1. Переопределение метода потомка перегруженным методом базового класса (с использованием объявления using) 262

Пример 6.2. Делегирование методов (графический редактор «Окружности и квадраты») 264

Пример 6.3. Простые свойства (класс Целое число) 269

Пример 6.4. Свойства-массивы (класс Динамический массив) 270

Пример 6.5. Совместная обработка исключений различных типов 282

Пример 6.6. Статические, виртуальные и динамические полиморфные методы 285

Пример 6.7. Разработка VCL-совместимого класса для реализации главного окна приложения «Динамический массив» 290

Пример 6.8. Перекрытие виртуальных методов в С++ классах и VCL-совместимых классах 296

Пример 6.9. Инициализация полей при конструировании объектов VCL-совместимых классов 297

Пример 6.10. Вызов виртуального метода из конструктора 299

Учебное издание

ИНФОРМАТИКА В ТЕХНИЧЕСКОМ УНИВЕРСИТЕТЕ

Иванова Галина Сергеевна
Ничушкина Татьяна Николаевна
Пугачев Евгений Константинович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Редактор *Н.Е. Овчеренко*
Художники *С.С. Водчиц, Н.Г. Столярова*
Корректор *Л.И. Малютина*
Компьютерная верстка *И.Ю. Бурова*

Изд. лиц. ЛР № 020523 от 25.04.97

Подписано в печать 27.03.2001. Формат 70x100/16. Печать офсетная. Бумага офсетная
Гарнитура «Таймс». Усл. печ. л. 26. Уч.-изд. л. 26,28. Тираж 3000 экз. Заказ 1459

Издательство МГТУ им. Н. Э. Баумана,
107005, Москва, 2-я Бауманская, 5.

Отпечатано в соответствии с качеством предоставленного оригинал-макета
в ППП «Типография «Наука» 121099, Москва, Шубинский пер., 6